Tolerância a falhas em sistemas de agentes móveis

Alex Jeukens

Tolerância a Falhas em Sistemas de Agentes Móveis

Alex Jeukens

Tolerância a Falhas em Sistemas de Agentes Móveis

Prof. Dr. Célio Cardoso Guimarães
Prof. Dr. Edmundo R. M. Madeira
Prof. Dr. Flávio Morais de Assis Silva
Prof. Dr. Ricardo de Oliveira Anido (orientador)

Jeukens, Alex

J524t Tolerância a falhas em sistemas de agentes móveis/Alex Jeukens – Campinas, [S.P. :s.n.], 2003.

Orientador: Ricardo de Oliveira Anido.

Dissertação (mestrado) – Universidade Estadual de Campinas. Instituto de Computação.

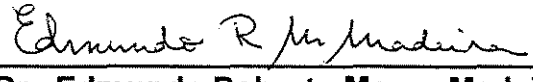
1. Redes de computação – Protocolos. 2.Tolerância a falhas (Computação). 3.Sistemas operacionais distribuídos (Computadores).
I. Anido, Ricardo de Oliveira. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 04 de dezembro de 2003, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Flávio Morais de Assis Silva
UFBA

Prof. Dr. Célio Cardoso Guimarães
ICMC - USP



Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Prof. Dr. Ricardo de Oliveira Anido
IC - UNICAMP

Tolerância a Falhas em Sistemas de Agentes Móveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Alex Jeukens e aprovada pela Banca Examinadora.

Campinas, 04 de Dezembro de 2003.

Prof. Dr. Ricardo de Oliveira Anido.
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Abstract

Mobile agent is a process capable of roaming autonomously through the network, executing operation locally to a host. Agents can be employed to perform some traditional tasks. In order to enforce dependability in this new paradigm we present a protocol to coordinate a replicated agent system capable of tolerating fail-stop and communication faults.

Sumário

Agente móvel é um processo que se desloca autonomamente através da rede, executando operações locais à máquina hospedeira. Agentes constituem uma solução alternativa ao paradigma cliente-servidor. A autonomia do agente móvel, entretanto, faz necessária a revisão dos conceitos de tolerância a falhas a fim de garantir confiabilidade em sua operação, evitando que a parada do agente resulte em inconsistências no sistema. Este trabalho apresenta um protocolo para tolerar falhas do tipo falha-e-pára e de comunicação.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1 - Introdução

Um agente móvel é um programa de computador que executa uma tarefa autonomamente em favor de um usuário e pode deslocar-se através de uma rede de sistemas heterogêneos [Ple02]. Por autonomamente entende-se que o agente pode decidir quais máquinas visitar e quais operações executar em cada uma. Um sistema de agentes móveis tolerante a falhas apresenta requisitos adicionais àqueles necessários para que um sistema tradicional (ex: clientes-servidor) possa tolerar falhas. Quando replicação é utilizada, o caráter ativo dos agentes exige que o grupo seja coordenado, para evitar a reexecução de uma mesma operação. O estudo deste problema, que é conhecido como o da garantia da unicidade da execução da tarefa, é o objetivo desta tese que será alcançado com um novo protocolo de coordenação de agentes, apresentado no capítulo 5.

Portanto neste trabalho será abordado o problema de garantir a sobrevivência de um grupo de agentes ao longo de uma computação longa, que envolverá a visita e a realização de operações em várias máquinas. O grupo de agentes poderá ainda manipular dados no contexto de uma transação, e desta maneira será conferida um caráter atômico à computação.

No presente capítulo serão apresentadas algumas características e vantagens da utilização de agentes móveis. Também serão listadas algumas aplicações de agentes móveis. No capítulo 2 o conceito de agente móvel é discutido com maior profundidade. No capítulo 3 é apresentada uma revisão bibliográfica sobre transações. Alguns trabalhos (ex: [AS99]) abordaram a questão da integridade dos dados manipulados pelos agentes de maneira integrada com a questão da recuperação do agente. A tendência moderna [Ple02], entretanto, é separar as duas questões para permitir a utilização de um modelo de transação qualquer em conjunto com o protocolo que garanta a recuperação do sistema de agentes. No capítulo 4 são apresentados os trabalhos sobre a recuperação de agentes. No capítulo 5 é apresentado com detalhes o protocolo para coordenar os agentes móveis. O capítulo 6 conclui esta dissertação de mestrado.

Agentes móveis possuem as seguintes características:

Diminuição do tráfego na rede: O fato de um agente poder recolher e processar informações localmente pode representar vantagens em relação ao envio de todas as informações para um servidor central. Por exemplo, em uma aplicação de gerenciamento de informações de centrais telefônicas é necessário gerar relatórios a partir dos dados sobre as chamadas armazenadas na central. Ao invés de enviar uma massa de dados para ser processada em uma máquina remota, é conveniente enviar o código (agente) para realizar a análise na própria máquina e retornar apenas com um relatório.

Pouca sensibilidade a partições de rede: falhas que isolem o agente de seu nó de origem não necessariamente paralisarão a execução da tarefa, caso não seja necessário mudar de partição.

Capacidade de integrar sistemas legados: quando há restrições sobre quantidade de memória, é difícil escrever um código que implemente um grande número de funções. Para contornar a limitação de memória, é possível definir

agentes que executem tarefas específicas e carregá-los no sistema de acordo com a tarefa do momento.

Adaptação dinâmica: agentes podem aproveitar sua autonomia e mobilidade para ajustar sua operação frente a mudanças do ambiente, por exemplo, migrando para um servidor livre após detectar sobrecarga do atual.

Capacidade de agregar requisições: agentes podem carregar várias requisições de um protocolo síncrono e executá-las localmente em um servidor remoto. Desse modo evita-se transmitir várias mensagens através da rede, sendo esta uma forma de utilizar agentes para introduzir assincronismo em um protocolo síncrono.

É possível utilizar agentes móveis para desenvolver aplicações distribuídas que, tradicionalmente, seriam organizadas segundo o paradigma cliente-servidor. Com freqüência é observado um aumento de desempenho, resultado da redução da comunicação entre o lado cliente e o lado servidor [PB95].

A seguir são apresentadas algumas aplicações do paradigma de agentes móveis:

1.1 Controle de aplicações de tempo real

Controlar remotamente um dispositivo exige a alocação de certa porcentagem da capacidade de comunicação, durante todo o período de controle. Além de dispendioso, esse controle pode ser inviável quando a distância entre as localidades é muito grande (ex: controlar a sonda marciana Sojourner a partir da Terra; a distância entre a Terra e Marte é, aproximadamente, dez minutos na velocidade da Luz). Enviar um agente para controlar o dispositivo no local reduz a comunicação e o tempo de resposta.

1.2 Monitoramento e recuperação de valores

Um agente lê constantemente valores como preço de ações na bolsa e retorna ao seu usuário caso ele ultrapasse um determinado valor. Isso elimina mensagens periódicas para informar ao interessado as variações no preço.

1.3 Computação móvel

Hoarding é a operação que precede a desconexão de um *host* móvel que consiste na transferência para o aparelho de toda a informação que será utilizada durante o período em que ele estiver desconectado. Essa coleta de informações pode ser realizada por agentes que lêem a informação para depois migrar para o *host* móvel.

1.4 Redes ativas

Ao contrário das redes tradicionais que se limitam a realizar roteamento, as redes ativas permitem que códigos de programas sejam carregados por pacotes para serem executados em nós intermediários, realizando computações e modificando o conteúdo de outros pacotes. As características de redes ativas podem ser aproveitadas em ambientes de telecomunicação, gerenciando serviços em pontos específicos da rede.

Uma rede ativa virtual pode ser descrita por um grafo e pode ser implementada através de agentes móveis que atuam como serviços móveis. O agente percorre a rede coletando informações (por exemplo, a taxa de transferência em dois pontos da rede) e, em seguida, passa a atuar ativamente alterando a configuração dos aparelhos, a fim de otimizar o funcionamento da mesma.

1.5 Manutenção e atualizações de software

Agentes podem ser utilizados para eliminar a necessidade de uma base centralizada de informações. Cada sistema é analisado individualmente por um

agente que determina a necessidade e possibilidade de atualização a partir da descrição de uma configuração básica. É possível programar a atualização de grupos de máquinas, de modo a satisfazer necessidades particulares.

A facilidade e simplicidade de se programar um agente para realizar a tarefa de atualização é o primeiro argumento em favor de sua utilização. A programação mais rudimentar é formada por uma lista de *softwares* e o respectivo local da instalação. E possível sofisticar a tarefa definindo compatibilidades entre *softwares* e deixando o agente decidir qual a melhor configuração. Elimina-se a base centralizada de informações e as mensagens de controle entre a máquina do administrador e as máquinas em atualização. É possível realizar a atualização em paralelo, não havendo sobrecarga de processamento na máquina do administrador porque o agente utiliza o processador da máquina em atualização.

Pode ocorrer que dois administradores desejem realizar atualizações simultaneamente, definindo listas concorrentes. Nesse caso, a tarefa de atualização pode apresentar um caráter transacional.

1.6 Comércio eletrônico

Um agente pode ser programado para representar seu usuário, pesquisando preços e produtos, realizando compras em *sites* de comércio eletrônico [Con98]. A cada passo, o agente realiza uma análise das ofertas e mantém o registro da melhor oferta em um objeto interno. Evita-se, assim, o transporte de informações desnecessárias pela rede.

Capítulo 2 - Sistemas de Agentes Móveis

Um dos mais antigos sistemas de agentes móveis que se tem notícia é o Telescript [Whi96] cujo nome referencia a linguagem interpretada em que os agentes eram escritos (Low Telescript). A escolha de uma linguagem interpretada é justificada pela identificação do conceito de interpretador com o de agência, no sentido de o interpretador controlar e intermediar o acesso do código (agente) aos recursos do sistema e pela facilidade de portar este interpretador. Logo que foi apresentada, JAVA [GM95] tornou-se a linguagem preferida para o desenvolvimento de sistemas de agentes, por apresentar bibliotecas (*packages*) para uso de rede e segurança (criptografia). CORBA é um conjunto de especificações que definem e padronizam a interação entre objetos distribuídos, implementado como um conjunto de bibliotecas que complementam e enriquecem a linguagem JAVA com conceitos como interceptadores, serviço de transações [TS01], qualidade de serviço, *secure socket layer* (SSL), controle de concorrência [CCS00]. Posteriormente, foram apresentados sistemas de agentes CORBA/JAVA como o Voyager, que representam, até a presente data, o estado da arte em sistemas de agentes móveis. Aglets influenciou a especificação CORBA para a interoperabilidade de sistema de agentes [MF98] (*Mobile Agent System Interoperability Facilities*). Essa especificação define os elementos de um sistema de

agentes, nomes de métodos e parâmetros, garantindo a interoperabilidade entre sistemas de agentes de fabricantes diferentes.

2.1 Conceitos

Os principais conceitos relacionados ao paradigma de agentes móveis serão discutidos nesta seção.

2.1.1 Mobilidade

Em geral, o estado de um programa qualquer é discriminado pelo conjunto de variáveis definidas em código, pelo valor das estruturas internas do sistema de execução e pelo valor de registradores do microprocessador.

Uma forma de implementar a migração de *threads* [SDR95] é definir um estado que satisfaça às seguintes propriedades:

- ser totalmente representado por variáveis definidas no código do agente;
- conter, de forma absoluta, a indicação do próximo estado.

Desta forma, a operação de migração pode ser definida pelo seguinte algoritmo:

1 Interromper a execução do *thread* `P1` em `C1`.
2 Instanciar em `C2` o *thread* `P2`, de código idêntico a `P1`.
3 Copiar o estado de `P1` em `P2`.
4 Destruir o *thread* `P1`.
5 Iniciar a execução do *thread* `P2`.

Figura 2.1 – Algoritmo para migração de agente.

De fato é possível alcançar qualquer outro estado a partir deste estado especial.

2.1.2 Agente

Um agente móvel é um programa autocontido, capaz de se deslocar autonomamente pela rede. [Ple02]

2.1.3 Agência

É um ambiente que intermedia e controla o acesso do agente a itens de informação. A autonomia de movimentação dos agentes pode implicar em riscos de segurança, caso não seja possível controlar seu acesso aos referidos itens. JAVA possui um sistema de controle de acesso denominado *sandbox* que é criado por meio de regras contidas em um arquivo chamado `java.policy`. As regras definem permissões de acesso. É possível restringir/conceder acesso baseado na URL (*Uniform Resource Locator*) de origem do agente, ou no usuário representado pelo agente. Pode-se aplicar a regra a um arquivo específico ou a um conjunto de diretórios.

2.1.4 Itens de informação

São recursos fixos disponíveis em uma determinada máquina, organizados de forma simples, como um arquivo texto, ou complexa, em um banco de dados.

2.1.5 Itinerário

É o conjunto de operações e dos locais de execução que compõem a tarefa do agente. Cada operação é formada por um conjunto de dados:

- a identificação do agente que executará a operação. É possível que um agente seja clonado, portanto é necessário discriminar se a instrução deverá ser executada pelo agente original ou um de seus clones;
- o local (agência) onde será executada;
- a ação (método, consulta ou programa que será executado).

Considere uma tarefa em que um agente deve atualizar o *driver* TCP de três agências de endereços: `local0`, `local1` e `local2`. A tarefa pode ser acelerada pela clonagem do agente no `local1`, para que o agente original possa atualizar o *driver* da agência no `local2`, enquanto o clone atualiza a agência do `local1`.

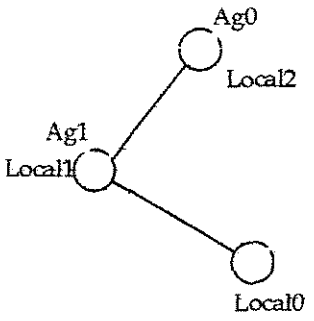


Figura 2.2 - Atualização simultânea em local1 e local2

Abaixo é representado um itinerário com a atualização simultânea em `local1` e `local2`.

Identificação do agente	Agência	Operação
0	local0	*update* TCPDriver
0	local1	*spawn*
1 (clone)	local1	*update* TCPDriver
0	local2	*update* TCPDriver

Tabela 2.1 - Exemplo de itinerário

2.1.6 Operações básicas

Em sistemas de agentes mais recentes, como o `Voyager` [VOY], é comum encontrar as seguintes operações:

- Migração (*hop*) entre máquinas distintas.

- Clonagem (*spawn*): é a criação de uma cópia do agente que difere apenas no ID (número de identificação). A operação de clonagem é tipicamente utilizada para criar paralelismo.

- Serviço de mensagem: permite a comunicação entre agentes e com recursos distribuídos.

2.2 Aglets

O Aglets é um *framework* para o desenvolvimento de sistemas baseados em agentes móveis. A classe que representa um agente é denominada Aglet e possui métodos que podem ser invocados em resposta a eventos preestabelecidos do sistema:

`OnCreation()`	Logo após a criação do agente.
`OnDisposing()`	Após sua destruição.
`onArrival()`	Após a chegada do agente a uma agência.
`OnDeparture()`	Imediatamente antes da partida do agente para uma nova agência.
`Run()`	Após o método `onCreation()` e após cada migração.
`handleMessage(Object obj)`	Chama a rotina para processar uma mensagem.

Tabela 2.2 - Operações básicas de um agente Aglets

Cada agente é executado em sua própria *thread* e pode trocar mensagens de forma assíncrona.

O sistema de agentes Aglets implementa o conceito de *sandbox*, que é um ambiente de execução de operações potencialmente perigosas ao sistema. Por exemplo, é possível impedir o programa de realizar operação de escrita no disco e

desta forma apagar arquivos indevidos. São definidos domínios contendo agências, autoridades que compreendem o autor do código, gerente do domínio e usuário do agente. Para cada combinação de domínio/autoridade é definido um conjunto de regras para autorizar ou proibir uma determinada ação como, por exemplo, restringir o acesso a um arquivo ou o estabelecimento de conexão em uma determinada porta.

2.3 A especificação MASIF CORBA

A especificação MASIF (*Mobile Agent System Interoperability Facilities Specification*) ajuda a estabelecer interoperabilidade entre plataformas de sistemas de agentes de fabricantes diferentes [MF98]. Cada sistema de agentes foi classificado e associado a um código de identificação. A especificação MASIF aborda quatro questões básicas:

- Gerenciamento de agentes: refere-se à criação e encerramento das atividades do agente, bem como sua suspensão e reinício.

- Acompanhamento do agente: consiste em determinar a localização dos agentes em um ambiente distribuído.

Capítulo 3 – Tolerância a falhas em Sistemas Distribuídos

Os sistemas computacionais são por natureza muito complexos e compostos por um grande número de componentes e subsistemas que interagem para realizar as tarefas desejadas. A maneira de tornar os sistemas mais confiáveis é implantar técnicas de prevenção e tolerância a falhas. Prevenção de falhas é a tentativa de evitar que falhas ocorram, eliminando-as ainda em fase de projeto. Em particular, a prevenção e a remoção de falhas não tornam os sistemas totalmente livres de erros, pois componentes físicos têm seu tempo de vida comprometido e podem vir a falhar. É nesse contexto que surge a necessidade de técnicas de tolerância a falhas. O objetivo das técnicas é manter o funcionamento correto e a disponibilidade do serviço, mesmo que falhas de *hardware* ou *software* ocorram.

Neste capítulo serão inicialmente apresentados conceitos básicos de tolerância a falhas, como replicação e sincronismo, a fim de preparar o leitor para compreender tal problema no contexto de agentes móveis. A seguir serão apresentados alguns modelos de transações. A definição de uma computação no contexto de uma transação atribui a qualidade de atomicidade a esta computação: ou a computação

inteira é completada ou nenhum dado é alterado. O caráter atômico evita alterações parciais em caso de falhas, isto é, a perda da integridade da informação.

Embora transações não sejam o foco deste trabalho, é importante que possam ser utilizadas em conjunto com as técnicas para proteger agentes, porque caso ocorram mais que `f+1` falhas, onde `f` é número máximo de falhas toleráveis, a computação será interrompida. Mesmo que a computação possa prosseguir através da substituição dos agentes falhos, pode ser necessário cancelar operações. Os primeiros trabalhos [JMS+98], [RS98], [AS99] sobre tolerância a falhas em agentes móveis tratavam as questões da recuperação dos agentes quando da presença de falhas e da integridade dos dados de forma conjunta. Porém estes problemas são de fato ortogonais [Ple02]. No capítulo 4 é apresentada uma análise dos trabalhos relacionados, separando, para cada trabalho, as técnicas relacionadas a cada um destes dois conceitos. A idéia é permitir a composição de qualquer modelo de transação com qualquer técnica para recuperar agentes.

3.1 Conceitos

Muitos termos são usados para descrever o funcionamento correto de um sistema. O objetivo desta seção é definir de forma precisa a terminologia utilizada neste trabalho. Um conceito importante é a robustez[1] que é a propriedade que garante a integridade de comportamento de um sistema e nos permite confiar nos serviços fornecidos por ele. Engloba os quatro conceitos seguintes:

Confiabilidade: Probabilidade de que um sistema permanecer ativo durante todo o tempo de sua missão, realizando corretamente a tarefa que lhe foi designada.

Disponibilidade: Probabilidade de que um sistema esteja apto a fornecer seus serviços num determinado instante.

---

[1] do inglês: *dependability*

Segurança de Uso[2] : Capacidade de um sistema de impedir situações de risco para a vida, a saúde, o patrimônio e o meio-ambiente.

Segurança de Acesso[3] : Propriedade de um sistema de impedir que pessoas não autorizadas tenham acesso aos dados que mantém e aos serviços que oferece.

3.2 Distribuição e replicação

Todas as técnicas de tolerância a falhas se baseiam em distribuição e redundância de elementos do sistema, ou seja, seus componentes [LA90]. A estratégia de redundância de componentes visa aumentar a disponibilidade e qualidade do serviço fornecido.

Redundância pode ser tanto de:

Componentes de *hardware* ou *software*: Utiliza-se mais de um componente para desempenhar uma mesma função.

Projeto do sistema: Se houver falhas de projeto, a manifestação de uma falha ocorrerá simultaneamente em todas as réplicas. Neste caso, é importante a diversidade de projeto dos elementos para aumentar a probabilidade de que falhas concorrentes não ocorrerão, garantindo a qualidade do serviço fornecido.

3.3 Exceções

O objetivo tolerância a falhas é prevenir que falhas e erros levem ao defeito do sistema; portanto, estratégias para lidar com erros e falhas são importantes.

Gerenciamento de exceções é uma técnica bem estruturada que tem sido incluídas em diversas linguagens de programação orientada a objetos modernos para auxiliarem a implantação de técnicas de tolerância a falhas. A manifestação de

---

[2] do inglês: *Safety*

[3] do inglês: *Security*

falhas leva a erros no sistema e, quando identificados, sinalizam sua presença através do levantamento de exceções. Quando uma exceção é detectada, o mecanismo de tratamento de exceções é responsável pela interrupção do processamento normal do componente e pela procura de uma rotina adequada para corrigir os erros causados pela manifestação da falha.

A implantação das técnicas de tolerância a falhas de modo que erros e falhas possam ser detectados e recuperados eficientemente consiste em três fases:

Detecção do Erro: Responsável por sinalizar o erro através da criação de uma exceção.

Isolamento e Avaliação do erro: Responsável por avaliar os componentes afetados pelos erros.

Recuperação do Erro: Constitui o emprego do tratador para exceção levantada de modo a recuperar o sistema para um estado livre de erros.

3.3.1 Detecção de erro

O ponto de início para estratégias de tolerância a falhas é a identificação da manifestação de falhas em forma de erros. O sucesso de qualquer sistema tolerante a falhas depende, efetivamente, da técnica de detecção de erros. Os mecanismos empregados na detecção de erros têm como objetivo o levantamento de exceções. O tratamento da exceção levantada nesta fase é responsabilidade das fases subseqüentes.

Existem quatro tipos de falhas:

Falhas de Omissão: caracterizam-se pela ausência temporária de respostas por parte de um determinado componente.

Falhas de Tempo: consistem no atraso na execução de uma tarefa (ex: envio de mensagem).

Falhas de Parada: O componente se torna inacessível. Através de um monitoramento do comportamento desse componente, é possível sinalizar a ocorrência do erro pelo levantamento de exceções.

Falhas Bizantinas: O componente passa a exibir um comportamento arbitrário e malicioso. O componente responde a todas as requisições que lhe são feitas, dando sempre a impressão de estar funcionando corretamente. Suas respostas, porém, são arbitrárias e podem vir a manifestar falhas em outros componentes do sistema, que também passariam a se comportar erroneamente. Todos os componentes inclusive de *hardware* estão sujeitos a manifestar falhas Bizantinas, ao invés de simplesmente pararem de funcionar.

A maioria das medidas adotadas para detecção de erros causados por falhas Bizantinas se baseia em checagem de respostas. Abaixo são descritos os tipos comuns de checagem que requerem a geração de informação adicional, como por exemplo através de um componente idêntico para produzir um segundo conjunto de dados independente:

Checagem de Réplicas: provê um dos mais poderosos e completos mecanismos para detecção de erros, mas também está entre as técnicas mais caras em termos da redundância requerida.

Consiste na replicação de elementos críticos do sistema e adição de um novo elemento que compara os resultados obtidos, o avaliador. Esta técnica é geralmente empregada para detecção de falhas de *hardware*, pois os softwares podem estar sujeitos a falhas de projeto que afetam todas as réplicas.

Checagem de Tempo: Se a especificação do componente inclui informações sobre o tempo necessário para o fornecimento do serviço, uma exceção de *timeout* pode indicar a falha daquele componente. Aplica-se tanto a *hardware* quanto a software. Mas, enquanto a exceção de *timeout* indica a falha em um componente, a sua ausência não garante o funcionamento correto do sistema. Desse modo, deve ser sempre empregada juntamente com outras técnicas.

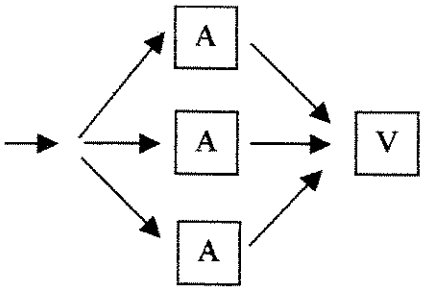


Figura 3.1 - Redundância Modular

Cheque do Reverso: Alguns sistemas possuem um mapeamento complexo entre as entradas e suas correspondentes saídas. Em outros sistemas, a relação entre entrada e saída é uma função direta e, assim, a saída de um componente pode ser usada para calcular qual foi a entrada que lhe deu origem. Esta entrada calculada é comparada com a entrada real para checagem da correta execução do sistema.

Cheque de Redundância: Um exemplo é o cheque de paridade que consiste em associar um ou mais *bit(s)* de paridade a um conjunto de *bits*, tal que a informação de paridade possa ser confrontada com a informação da mensagem. É usado para detectar problemas no armazenamento ou transmissão de dados. Se comparado com as outras técnicas, em termos de redundância requerida, pode ser visto como uma solução eficiente e econômica.

3.3.2 Isolamento e avaliação do dano

Quando um erro é detectado, muitos outros componentes já podem ter sido influenciados por ele. Isso acontece porque existe um atraso entre a manifestação da falha e a detecção de suas conseqüências errôneas. Durante esse período, muita informação incorreta pode ter sido disseminada entre os elementos do sistema, levando à ocorrência de outros erros que ainda não tenham sido detectados. Dessa

forma, antes que se tente recuperar o erro, é necessário avaliar a extensão do dano causado ao sistema.

Estratégias para avaliação de danos baseiam-se nas estruturas presentes nos sistemas operacionais que ajudam a confinar o erro causado por uma falha. As técnicas de isolamento de danos sempre envolvem decisões subjetivas sobre o fluxo de informações. Um conceito muito usado é o de ações atômicas. A atividade de um grupo de componentes constitui uma ação atômica, se não existir interação entre os elementos desse grupo e o resto do sistema, durante a execução da referida ação. O paralelismo e as ações aninhadas dificultam o isolamento do erro e, conseqüentemente, a avaliação dos danos. Enquanto isso, a distribuição e descentralização, com múltiplos processadores autônomos, são importantes para evitar uma disseminação do erro embora dificultem o planejamento de estratégias para avaliação dos danos.

Na prática, as técnicas de avaliação de danos são os aspectos mais incertos e incompletos da tolerância a falhas e estão intimamente relacionadas com as técnicas de detecção e recuperação de erros.

3.3.3 Recuperação do erro

Após a identificação do erro e verificação do dano por ele causado, é necessário eliminá-lo através das técnicas de recuperação que objetivam levar o sistema a um estado livre de erros. A recuperação do erro é responsabilidade do tratador (*handler*) da exceção sinalizada.

Duas técnicas podem ser usadas para eliminar erros de um estado do sistema: (i) por *avanço* e (ii) por re*trocesso* [AK86]. A técnica de por *avanço* manipula o estado corrente para produzir um novo estado livre de erros. Ela é bastante eficiente, pois realiza alterações limitadas e específicas em estados do sistema; entretanto, para que isto seja possível, é preciso saber de antemão quais os estados afetados por tal falha. É uma técnica específica para sistemas onde o dano causado por uma falha

pode ser previsto e, naturalmente, o seu sucesso depende da eficiência das técnicas de identificação e isolamento de erros. A segunda técnica, por re*trocesso*, tenta restaurar um estado anterior do sistema que esteja livre de erros. Esta técnica não se baseia na previsão de informações de estados afetados pelas falhas e, por isto, é mais genérica e facilmente proporcionada por um mecanismo.

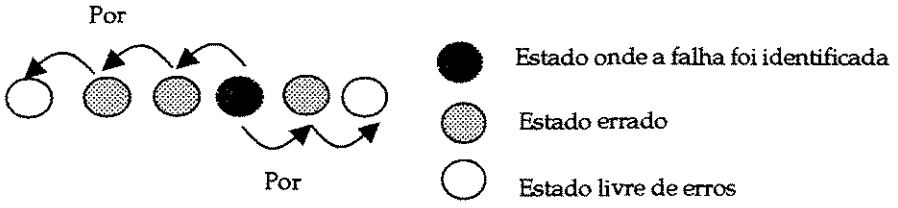


Figura 3.2 - Técnicas por *avanço* e por *retrocesso* para recuperação dos erros de um sistema

Exatamente porque a técnica por re*trocesso* possui os benefícios mencionados anteriormente, é extremamente poderosa e bastante empregada na prática. *Forward*, porém, é muito mais econômico, pois atinge diretamente os componentes afetados, enquanto por re*trocesso*, além de agir sobre todo o sistema, demanda recursos extras para armazenamento de informações necessárias para a recuperação do estado consistente. Além disso, existem componentes que não podem ser restaurados a estados anteriores devido a restrições de custo ou físicas.

3.3.4 Tratamento da falha e continuação do serviço

Embora a técnica de recuperação de erros tenha retornado o sistema para um estado livre de erros, ainda são necessárias outras técnicas que permitam que o sistema continue funcionando. O importante é garantir que a mesma falha não se repetirá imediatamente, retornando o sistema para um estado instável. Manifestações freqüentes de uma mesma falha podem forçar o sistema a falhar, seja porque as conseqüências da falha vão se agravando, seja porque o sistema está

ocupado em restaurar estados estáveis, não tendo condições de fornecer o serviço de sua responsabilidade.

A técnica de tratamento de falhas pode ser dividida em duas etapas:

- Localizar a falha;

- Reparar a falha ou reconfigurar o sistema.

Existe um sério problema em identificar a falha que precisa ser tratada. A detecção do erro não necessariamente serve para identificar a falha que o causou, pois mais de uma falha pode gerar erros do mesmo tipo. A presença da falha é indicada por uma exceção e, em muitos sistemas, nenhuma informação adicional é fornecida para auxiliar a identificação das falhas. Nestes casos, o sistema só poderá se recuperar se a exceção levantada possuir certas características que permitam identificar o componente e a falha responsável por sua manifestação.

Quando a falha for localizada, um ou mais componentes do sistema serão considerados causadores e, para prevenir que voltem a causar problemas, eles devem ser reparados. Técnicas para reparos em sistemas são essencialmente baseadas na reconfiguração:

1. **manual:** quando todas as ações são realizadas por um agente externo ao sistema (usualmente humano);
2. **dinâmica:** quando as ações são realizadas pelo sistema, mas em resposta a instruções externas;
3. **espontânea:** quando as ações são inicializadas e realizadas pelo próprio sistema

Devido ao custo e dificuldade envolvidos em construir sistemas capazes de realizar efetivos reparos em suas próprias estruturas, técnicas de reconfiguração dinâmica e espontânea são encontradas apenas em sistemas onde a intervenção manual é inacessível ou onde o atraso proporcionado pelos métodos manuais é inaceitável.

Um método simples para a reconfiguração do sistema é incluir pontos fixos a partir dos quais ele pode recomeçar a execução. Por outro lado, alguns sistemas necessitam de informações da computação atual do sistema. Para sistemas de *hardware*, uma nova tentativa é freqüentemente considerada (`retry`). Para sistemas de software, se o tratador da exceção obteve sucesso na recuperação da falha, o controle pode ser retornado para a localização do sistema onde a falha ocorreu e este pode continuar a prestar seus serviços. Se o tratador não obteve sucesso, uma exceção de defeito deve ser sinalizada para que um outro componente tente tratar essa nova exceção.

Suponha que um comando `C` tenha lançado uma exceção para quem solicitou sua execução. Várias ações são possíveis de serem implementadas, dependendo do contexto de manifestação da falha: `Retry` repete a chamada a `C`. Deve-se considerar apenas se a exceção levantada for conseqüência de uma falha transiente ou se o tratador conseguiu eliminar os erros inseridos no sistema. O comando `Continue` executa o próximo comando ignorando a exceção levantada. Deve-se considerar apenas se o comando `C` era de extrema importância para o sistema. `Exit` termina a execução e `goto` especifica para onde o controle deve ser transferido.

3.4 Replicação

Manter várias cópias físicas de um mesmo dado distribuído em diferentes localidades é uma forma de se beneficiar de todas as propriedades de uma computação distribuída, tais como melhor desempenho, maior tolerância a falhas, distribuição de carga e extensibilidade. Mas a replicação torna o acesso aos dados muito mais custoso do que em sistemas não replicados, por necessitar garantir a consistência das informações de todas as réplicas. Assim, enquanto a replicação oferece todos os benefícios de um sistema distribuído, ela exige um protocolo de

controle - em geral não simples de implementar - para sincronizar todos os acessos às cópias e manter o estado consistente.

3.4.1 Transparência

Um requisito chave quando dados ou serviços são replicados é a transparência de replicação. O cliente não está ciente da existência de várias cópias, manipulando a informação como se existisse um único item.

3.4.2 Consistência

Um outro requisito importante para replicação de dados é a consistência das informações armazenadas em todas as réplicas. Normalmente não é aceitável que diferentes clientes obtenham resultados diferentes quando consultam a mesma informação.

A noção de consistência considerada em sistemas replicados contém dois aspectos: consistência mútua e consistência interna. Consistência mútua refere-se ao fato de que as cópias representam um mesmo dado lógico e os usuários devem perceber consistência dos dados mesmo que exista uma eventual disparidade entre as cópias.

A maior parte dos protocolos de atualização de réplicas se empenha em manter a consistência mútua entre as réplicas, considerando que a consistência interna é mantida por algum protocolo de controle de concorrência.

3.4.3 Ordenação

A ordem em que as requisições são processadas em diferentes réplicas é uma questão muito importante, não apenas porque geralmente é necessário que uma ordem particular seja obedecida para obter um resultado correto, mas também porque a necessidade de se ordenar as mensagens demanda recursos caros.

Primeiro, o processamento de uma mensagem pode ser atrasado porque uma requisição anterior ainda não foi processada. Como foi mencionado anteriormente, um protocolo para garantir uma ordem em particular pode ser muito caro de se implementar devido ao número de mensagens que serão transmitidas.

Existem basicamente dois tipos de ordenação: física, causal e total.

No processamento físico utiliza-se tempo de relógio (*timestamp*) para registrar o momento de execução de uma dada ação. Em um sistema distribuído é necessário sincronizar os relógio das máquinas que o integram. Este *timestamp* é utilizado para ordenar as ações; é interessante ressaltar que duas ações podem ocorrer simultaneamente. Sob um processamento fisicamente ordenado, se `r1` e `r2` são requisições, então, ou `r1` é processado antes de `r2` em todas as réplicas, ou `r2` é processado antes de `r1` em todas as réplicas.

Na ordenação causal é mantida a ordem relativa entre operações que possuem uma relação causa-efeito. Em geral, nem todos os pares de mensagens necessitam ser processados em uma determinada ordem. Requisições `r1` e `r2` cujo efeito de serem processadas em uma ordem (`r1r2`) é o mesmo de serem processadas em uma ordem inversa, (`r2r1`) são chamadas operações comutativas. Exemplificando, duas operações de leitura são comutativas.

Na ordenação total são estabelecidos critérios para estabelecer uma ordem absoluta entre ações.

3.4.4 Controle das réplicas

Protocolos para gerência de sistemas replicados são geralmente divididos em duas classes. Na primeira, chamada replicação ativa ou abordagem de máquinas de estados, o controle do sistema é descentralizado. A segunda classe de protocolos é chamada de abordagem passiva ou *primary-backup*. Nela o controle fica totalmente centralizado em uma réplica primária e todas as outras funcionam apenas como cópias de segurança (*backups*). Existe ainda uma abordagem

intermediária entre a ativa e a passiva, a abordagem preguiçosa (*lazy*). Em sistemas deste tipo, clientes comunicam com apenas uma réplica, a primária sob o seu ponto de vista, mas cada cliente possui um primário diferente. Tal abordagem é conhecida como preguiçosa por realizar a comunicação entre as réplicas depois de já ter atendido o cliente e se for estritamente necessário.

Em protocolos deste tipo uma das cópias é designada a ser primária (*primary*), sendo responsável por controlar todas as requisições que alteram o estado da réplica. Uma operação de escrita é sempre realizada no primário que propaga o novo valor para as outras cópias. Operações de leitura podem ser processadas por quaisquer réplicas ativas.

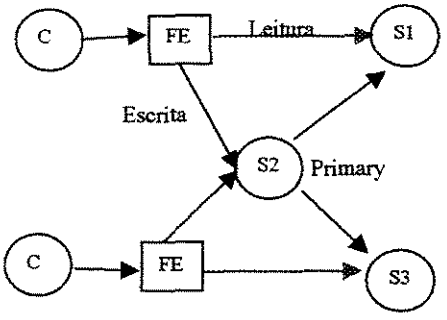


Figura 3.3 - *Primary-Backup*

Se o primário falhar, um *backup* é designado para ocupar o cargo de primário. O cliente é informado pelo sistema do novo primário e, a partir desse momento, ele irá direcionar suas requisições a outro servidor.

A abordagem *primary-backup* pode apresentar variações descritas a seguir:

(i). *Lazy*

A técnica de replicação preguiçosa (*lazy*) é uma maneira de se preservar a consistência dos estados replicados e proporcionar um bom desempenho através do uso de ordenação causal [AL97].

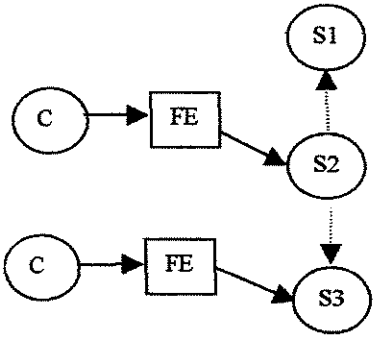


Figura 3.4 - *Lazy Replication*

Nesta arquitetura uma requisição é feita apenas a uma réplica, a primária sob o ponto de vista daquele cliente, e a atualização das demais é feita através de mensagens trocadas entre elas (*gossip*). O sistema é baseado em ordenação causal, uma réplica, e assim mesmo que não totalmente atualizada, pode ser capaz de

atender certas requisições de um cliente, ou seja, a réplica processou todas as operações que precedem logicamente a requisição. *Gossips* são enviadas apenas quando estritamente necessárias, ou seja, quando a réplica não consegue atender às requisições de seu cliente por estar desatualizada. Daí deriva o nome da técnica "replicação preguiçosa".

(ii). *Cold*

Abordagem passiva de replicação onde apenas a réplica primária executa as requisições feitas ao grupo. O estado do primário é armazenado em um arquivo de *log* para ser carregado em uma réplica *backup*, quando o primário falhar.

(iii). *Warm*

Abordagem passiva de replicação onde apenas a réplica primária executa as requisições feitas ao grupo e seu estado, armazenado-as em um arquivo de *log*, repassado-as aos *backups* periodicamente.

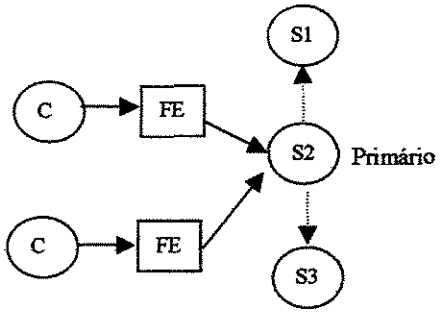


Figura 3.5 - *Warm* e *Cold*

A abordagem ativa ou de máquinas de estados, como também é conhecida, é um método geral para implementar serviços, através da replicação de servidores, e coordenar as interações entre clientes e réplicas. Uma máquina de estado consiste em variáveis de estado que reproduzem seu estado, e comandos que alteram seu estado. Cada comando é implementado por um programa determinístico. As saídas de uma máquina de estado são completamente determinadas pela seqüência em que as requisições são processadas, independentemente do tempo e de qualquer outra atividade do sistema.

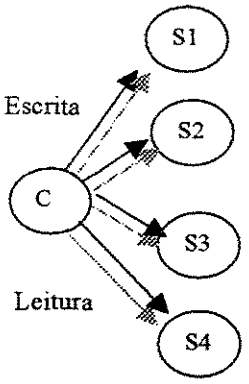


Figura 3.6 - Ativa

Uma versão tolerante a falhas de uma máquina de estados pode ser implementada replicando tal máquina e executando uma réplica em cada um dos processadores de um sistema distribuído.

Garantindo que cada réplica está rodando em um processador não faltoso e que cada uma delas estará executando ***todas*** as tarefas e ainda na ***mesma*** ordem podemos afirmar que elas produzirão o mesmo resultado. Mas, se assumirmos que uma falha pode afetar no máximo um processador, conseqüentemente uma única máquina de estados, então combinando as saídas das máquinas de estados replicadas podemos obter uma saída para uma máquina tolerante a falhas.

(i). Votação

Muitos dos protocolos existentes para controle de réplicas seguem a estratégia de permitir que operações de leituras e escritas sejam executadas ***apenas*** em um conjunto de cópias chamado *quorum*. Um *quorum* para duas operações conflitantes deve possuir, pelo menos, uma réplica em comum; ao se utilizar número de versão é possível que operações de leitura possam sempre escolher a cópia mais

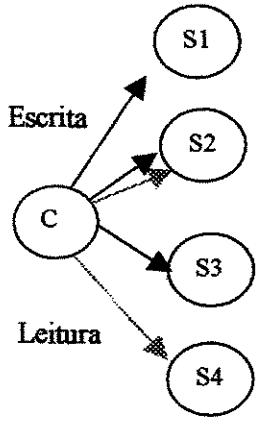


Figura 3.7 - *Votação*

recente e que duas operações de escrita irão ocorrer independentemente, excluindo assim a possibilidade das cópias divergirem. O número de mensagens trocadas pelas réplicas é proporcional ao tamanho do *quorum*. Esta arquitetura tenta minimizar o custo de todas as réplicas executarem todas as mensagens, proporcionando a mesma qualidade dos serviços da abordagem ativa.

(ii). Votação com testemunhas

Este protocolo é uma variação do anteriormente apresentado onde algumas cópias não contêm nenhuma informação, apenas o número de versão atual [PA86]. Para ratificar uma operação é necessária a aprovação da maioria das cópias (testemunhas ou não). O resultado porém, é gravado apenas nas réplicas, sendo as testemunhas atualizadas somente com a indicação do número da operação. A votação com testemunhas permite a redução do espaço de armazenamento, assim como o tempo da operação de escrita. Por outro lado, reduz-se a disponibilidade da informação.

(iii). Grupo de processos

Um grupo de processos é uma coleção de processos que cooperam para alcançarem um objetivo comum ou para processarem várias informações. Alguns sistemas replicados fazem uso de tais grupos para disponibilizarem serviços tolerantes a falhas de maneira cooperativa. Em sistemas distribuídos, *multicast* é o mecanismo de transmissão de mensagens de um processo para vários processos destinos.

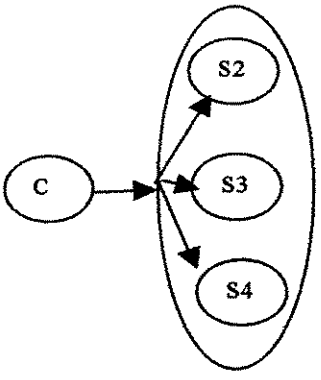


Figura 3.8 - Grupo de Processos

A comunicação de grupos aumenta a eficiência e conveniência da comunicação porque:

- proporciona um alto nível de abstração de comunicação;
- esconde da aplicação a coordenação interna do grupo como, por exemplo, mudanças nos membros dos grupos.

3.4.5. Replicação com agentes móveis

Como foi explicado na seção 2.1.5 o itinerário é formado por um conjunto de operações, cada uma a ser executada em uma máquina pré-determinada. Chama-se passo à execução de um subconjunto de operações da tarefa correspondente a uma determinada máquina acrescida da migração do agente para a próxima máquina.

Os agentes replicados podem ser organizados em duas configurações gerais:

Replicação Disjunta: um novo conjunto de nós é escolhido a cada passo para comportar as réplicas.

Replicação Linear: alguns nós utilizados no passo atual também farão parte do novo passo, normalmente aqueles recém visitados pelo agente, o que reduz o esforço de preparação do passo, reduzindo o tráfego de informação na rede.

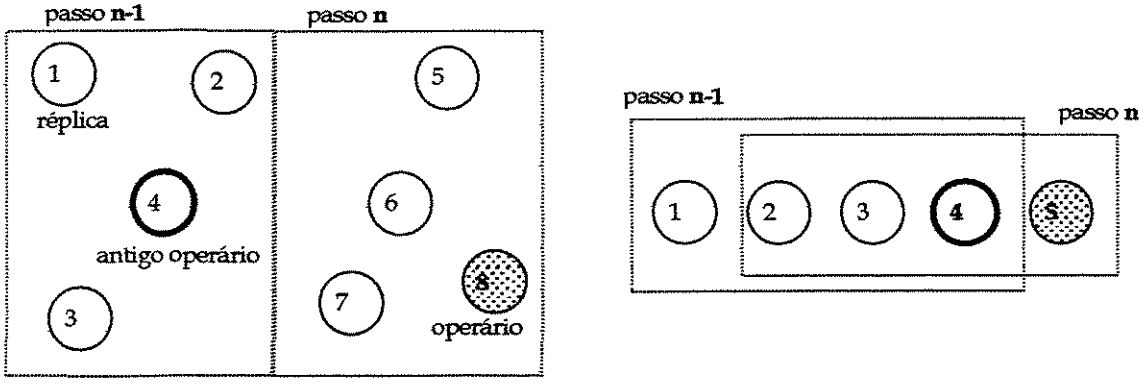


Figura 3.9 - Exemplo de Replicação Disjunta e Replicação linear

Se a política de incluir somente nós já visitados for utilizada a partir do primeiro passo, o sistema será executado com menos cópias que o definido como ideal, sendo maior a probabilidade do sistema bloquear, neste caso. Uma possível solução é utilizar replicação disjunta no primeiro passo e replicação linear a partir

daí, de forma que o sistema migre progressivamente para a configuração linear. A figura 3.9 exemplifica os dois tipos de replicação.

3.5 Transações

Tradicionalmente, define-se transação como uma estrutura de controle que envolve os conceitos ACID [GR93]:

Atomicidade - todas as operações são completadas com sucesso ou nenhuma é realizada;

Consistência – o sistema sempre se encontra em um estado correto: caso ocorra uma falha, ele é retornado a um estado correto anterior ou adiantado para outro estado diferente daquele inicialmente desejado;

Durabilidade – os efeitos das operações permanecem até a realização da próxima operação;

Isolamento – alterações associadas a uma transação são visíveis somente após a conclusão da transação.

Algumas aplicações não tradicionais apresentam requisitos adicionais como cooperação entre as transações, utilização de transações longas, de transações com estrutura complexa e autonomia entre bancos de dados. O modelo de transações tradicional que se baseia em serialização, não satisfaz esses requisitos, tornando necessária a introdução de novos modelos de transações, cada qual com estrutura particular, critérios de correção e mecanismos de recuperação.

3.5.1 Compensação

Compensação é um conceito utilizado para cancelar os efeitos de uma transação, que ainda não tenha sido finalizada, visíveis para outras transações [GR93]. A transação que viu os efeitos da primeira é dita dependente. É essencial

que tais efeitos possam ser anulados independentemente da execução da transação dependente, cuja execução deve poder ser preservada.

Compensação consiste na anulação semântica de efeitos de uma transação de forma a parecer que nunca foram realizados. Não é necessário que o banco de dados retorne ao estado que precedeu a transação.

A compensação de uma transação `t` é realizada por outra transação chamada compensatória, sendo a primeira chamada de "transação compensada". Uma vez que a transação compensatória é utilizada para estabelecer consistência, é obrigatório que seja realizado seu *commit*. Em [GMS87] considera-se possível compensar uma transação, quando ela pode ser decomposta em subtransações que possam ser intercaladas por qualquer outra transação.

Em termos gerais, após o *commit* de uma transação (que será) compensada, transações dependentes não devem alterar o estado do banco de forma a impossibilitar o *commit* da transação compensatória.

Seja `t` uma transação compensada, `tc` sua transação compensatória e `dep(t)` o conjunto de transações dependentes de `t`. Caso todas as operações de `dep(t)` sejam comutáveis com as operações de `tc` e `tc` realmente desfaça os efeitos de `t`, então é possível executar `tc` sem que esta última influencie os efeitos de `dep(t)`. É necessário analisar cuidadosamente a aplicação para determinar a possibilidade do uso de transações compensatórias.

3.5.2 Transações aninhadas

No modelo de transações aninhadas, uma transação pode ser formada por outras recursivamente, formando-se uma árvore de transações [Mos81]. As transações folhas, também chamadas de *flat* se desencadeiam como as definidas no modelo tradicional. Transações aninhadas contribuem para a modularização da aplicação, permitindo a definição de tratadores de exceção específicos para cada subtransação. O fracionamento da transação global aumenta a concorrência e,

juntamente com a possibilidade da execução em paralelo de subtransações, resulta no aumento de desempenho do sistema.

3.5.3 *Savepoints* e *checkpoints*

A propriedade de atomicidade da transação *flat* impõe que, na ocorrência de uma falha, as operações já realizadas devam ser anuladas. O sistema é retrocedido ou avançado para um estado livre de falhas, o que representa uma perda, que pode ser minimizada através do registro do estado do sistema em determinados pontos, os *savepoints* [GR93]. É possível retornar o sistema para o *savepoint* mais recente ao invés do estado inicial. Os *savepoints* são utilizados para superar falhas temporárias e resolver problemas de concorrência. Caso sejam gravados em mídia não volátil, passam a ser chamados de *checkpoints*. Os *checkpoints* podem superar falhas que provoquem a parada do programa. Neste caso é necessário incluir todas as variáveis internas do programa.

3.5.4 Sagas

O modelo de Sagas [GMS87] se baseia em compensação, sendo integrado por um conjunto de n transações t_1, t_2 ... t_n que podem ser intercaladas por outras transações. A cada transação t_i, 1 <= i < n, é associada uma transação compensatória c_i. Em um sistema que utilize transações Sagas executa-se a seqüência:

t_1, t_2, ... t_n ou

t_1, t_2, ... t_j, c_j, c_2, c_1 (para um j | 0 <= j < n).

O conceito de Sagas pode ser estendido para permitir a execução paralela de componentes da transação.

3.5.5 Transações-S

Este modelo é adequado para um ambiente formado por bancos de dados com grande autonomia [Vei90]. Cada banco possui um conjunto de transações predefinidas que constitui uma interface operacional para o mesmo. Elas definem o conjunto de funcionalidades de cada base de dados e apresentam as propriedades ACID. A base do modelo é a consistência individual e a observância às restrições locais dos bancos.

3.5.6 *Contracts*

No modelo *contracts* [WR92] a transação é dividida em etapas, que apresentam invariantes de entrada e saída. Para a transação estar correta é necessário que satisfaça aos invariantes. Caso não sejam satisfeitos de imediato, é possível disparar ações para adequar o estado do banco. Também faz parte do modelo um *script* de fluxo de controle e requisitos de sincronização e de recuperação. O modelo enfatiza a necessidade de programar as etapas de forma independente do *script*.

Contracts é flexível na definição do fluxo de controle, admitindo paralelismo de execução, laços e permitindo a utilização de resultados de execuções anteriores. A recuperação é baseada em compensação através de uma ação compensatória por etapa ou para um conjunto de etapas. O modelo permite a execução de etapas em componentes do sistema que não dispõem de semântica transacional.

3.5.7 Transações Migrantes (*Migrating Transactions*)

Transações móveis [KR88] é um modelo desenvolvido para lidar com transações de longa duração e controlar a execução da transação, em um ambiente distribuído. Uma transação móvel é composta por um conjunto de subtransações

que podem trocar informações entre si. As transações móveis utilizam o mesmo conceito de invariante apresentado na seção anterior.

Cada subtransação está associada a uma ação compensatória, sendo possível a transferência de informação entre elas, porém, neste caso, não são definidos invariantes.

O fluxo de controle de uma transação é determinado por meio de conectores lógicos e consultas sobre os resultados da transação anterior.

Para executar uma transação móvel é necessário predefinir junto ao gerenciador local, a subtransação que será executada em cada componente. Ao término da transação, o gerenciador atual enviará o resultado ao gerenciador que executará a próxima transação.

3.6 Transações *Split*

Transações *Split* [PKH98] são formadas por um conjunto de transações aninhadas distribuídas. São adequadas para os requisitos de atividades abertas, isto é, aquelas que são processadas indefinidamente, e interagem com outras atividades concorrentes. Quanto maior o tempo de duração da transação, maior é sua vulnerabilidade e menor a concorrência devido à retenção de recursos. Para superar tais problemas, uma transação *Split*:

- permite realizar o *commit* de recursos que não serão mais alterados, liberando-os;
- serializa o acesso a recursos para todas as atividades.

A operação de divisão (*split*) produz duas novas transações (`A` e `B`) a partir da transação original `T`. A operação inversa, reunião (*join*), encerra a execução da transação `T`, agrupando seu resultado em uma nova transação `S`:

```
Split-Transaction(
```

```
    A: (ConjLeituraA, ConjEscritaA, ProcedimentoA).
    B: (ConjLeituraB, ConjEscritaB, ProcedimentoB))
Join-Transaction (S:TID)
```

A partir do momento em que ocorre a divisão, considera-se que a transação recém dividida está finalizada, existindo a garantia de que os recursos não serão mais alterados. São definidos novos conjuntos de dados, a saber, `ConjLeituraA`, `ConjEscritaA`, `ConjLeituraB` e `ConjEscritaB`, bem como os procedimentos `ProcedimentoA` e `ProcedimentoB`.

São observadas as seguintes propriedades:

```
1 ConjEscritaA ∩ ConjEscritaB = UltimaEscritaB
2 ConjLeituraA ∩ ConjEscritaB = vazio
3 ConjLeituraB ∩ ConjEscritaA = ConjCompartilhado
```

A primeira propriedade garante que `A` não sobrescreverá os resultados de `B`, mas permite que `B` sobrescreva os de `A`. A propriedade dois permite que `A` seja serializado antes de `B`, uma vez que `A` não viu nenhum dos resultados produzidos por `B`. A propriedade três diz que `B` conhece os resultados de `A`.

Uma vez que o sistema de banco de dados garante a seriabilidade de `T`, se `A` e `B` podem ser serializados um em relação ao outro, eles podem ser serializados em relação a qualquer transação.

Se o `ConjCompartilhado` e `UltimaEscritaB` são vazios, não há conflito de acesso entre `A` e `B`, sendo possível serializá-los em qualquer ordem (caso independente). Uma vez que `A` e `B` são independentes, ambos podem continuar sem restrições adicionais.

Se `ConjCompartilhado` ou `UltimaEscritaB` não são vazios, `B` sucede `A` devido às dependências de acesso a dados (caso serial). Nesse caso, cada objeto do `ConjCompartilhado` deve permanecer inalterado em `A` após a operação de divisão, ou `B` pode estar utilizando informação inválida. No caso serial, se `A`

abortar em algum momento, B deve ser abortado ou B estaria utilizando dados inválidos.

Na operação de reunião, T une-se à transação S para realizar o *commit*. T pode ser considerada uma subtransação de S, ou T e S transações filhas de outra transação pai.

É possível reaplicar a operação de divisão para as transações resultantes de uma divisão anterior, bem como estender o modelo para incorporar o conceito de transações aninhadas.

3.6.1 Transações *Kanguroo*

O modelo de transações *Kanguroo* [DHB97] foi elaborado a partir das transações *Split*. É introduzido o conceito de transações globais, chamadas de transação *Kanguroo*, que servem como base para definir as transações móveis, chamadas *Joeys*.

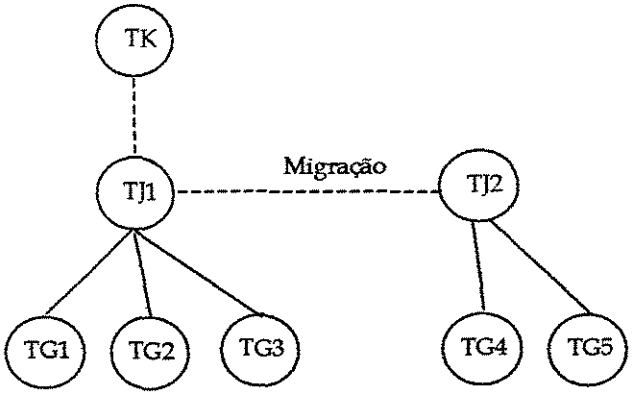


Figura 3.10 - Representação de uma transação *Kanguroo*

A transação *Kanguroo* é iniciada em um nó do sistema com a criação obrigatória do primeiro *Joey*. Um *Joey* contém a definição de uma transação local, existindo flexibilidade de escolha do modelo de transação local (*flat*, aninhado, etc). Para realizar operações em outro nó do sistema é necessário defini-las em um novo *Joey*,

criado a partir de uma operação de divisão (ver item 3.7.1), local ao referido nó. Os *Joyes* são relacionados entre si através de uma lista duplamente ligada.

Existem dois modos de operações com o *Joeys*:

compensation mode: neste modo, caso um *Joey* falhe, são executadas transações compensatórias para desfazer as operações de *Joeys* anteriores que já realizaram *commit*.

split mode: neste modo não há compensação para *Joyes* que já realizaram *commit*, caso ocorra uma falha. Apesar de não haver compensação, a transação atende ao critério de correção, uma vez que um *Joey* é criado através da operação de divisão.

Capítulo 4 - Trabalhos Relacionados

A liberdade do agente em poder migrar para um número indeterminado de agências durante a execução de sua tarefa, bem como desconhecimento da demanda da rede e do servidor onde a agência é executada, torna difícil determinar se uma eventual demora no retorno deste agente ao ponto de origem é normal ou é um indício de sua falha. Uma solução é utilizar replicação para superar a eventual falha de um agente. Entretanto, a característica do agente agir ativamente torna possível a reexecução de uma mesma tarefa quando uma das cópias não consegue determinar com exatidão a ocorrência de uma falha. É necessário que o grupo de agentes seja coordenado por meio de um protocolo para garantir que cada tarefa será executada uma única vez. Nos últimos anos, foram apresentados protocolos para coordenar o grupo de agentes; alguns assumiram o modelo falha e pára, outros adotaram um modelo que admite partições. Notou-se uma evolução desses protocolos, que se tornaram mais robustos, evitando que um único agente (coordenador) torne-se um ponto único de falha.

Em alguns trabalhos, foi abordada a questão da definição da tarefa como uma transação, entretanto esta é ortogonal à coordenação do grupo de agentes, sendo possível combinar modelos de transação com protocolos de coordenação.

4.1 Classificação dos protocolos de coordenação de agentes

Existem dois momentos para estabelecer a propriedade da unicidade da execução em um sistema de agentes móveis: controlar os agentes passo a passo ou ao final da execução. Realizar o controle passo a passo requer executar um protocolo que evite que o grupo de agentes se desagregue e permita que uma operação seja repetida por mais de um agente. Na segunda abordagem, somente ao final da tarefa será conhecida a série de agentes que obteve sucesso, assim é necessário que todos mantenham as travas sobre os recursos que alteraram até esse momento.

O agente que executa as operações é chamado operário enquanto o agente que centraliza e comanda o protocolo é chamado de coordenador.

Os protocolos de coordenação de agentes podem ser classificados segundo os seguintes critérios:

- Utilizar replicação para substituir o operário em caso de falha deste. Ou a tarefa é executada por um único agente ou é executada por múltiplos agentes.

- A coordenação do grupo pode ser realizada por um único membro ou pode contar com a participação de todos.

- O coordenador do processo decisório pode coincidir com o operário, isto é, a decisão pode ser justaposta à execução ou ser distribuída.

A figura abaixo mostra as possíveis combinações para os critérios apresentados:

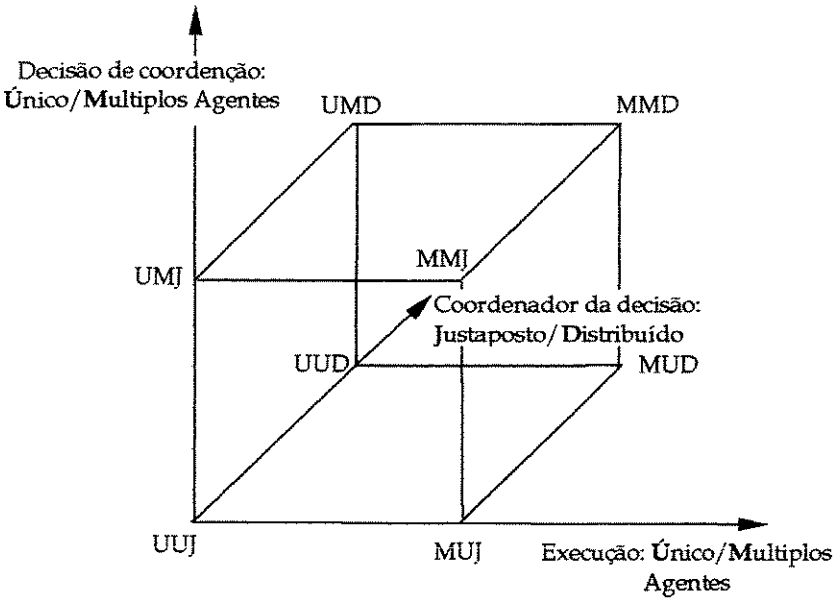


Figura 4.1 - Possíveis classificações para um protocolo de coordenação de agentes

Algumas combinações têm interesse apenas teórico; dentre os esquemas mais importantes na prática temos:

UUJ: Este esquema equivale a utilizar um único agente desprovido de mecanismo de tolerância a falhas. O agente executa as operações do passo e, quando terminar o passo, estará encerrado e o agente poderá prosseguir para a próxima agência definida em seu itinerário.

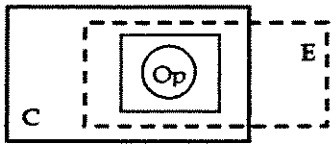


Figura 4.2 - Esquema UUJ

Os dois esquemas seguintes são adotados por muitos dos protocolos que realizam a coordenação dos agentes ao final do passo:

MUJ: Nesta configuração vários agentes podem executar as operações do passo, mas apenas um terá sucesso na execução do protocolo de finalização. Um exemplo é o protocolo apresentado em [RS98].

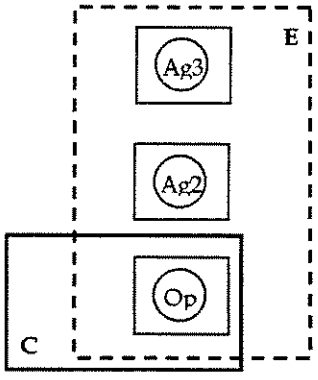


Figura 4.3 - Esquema MUJ

MMJ: É o mais interessante por garantir que o sistema não ficará bloqueado na eventual falha do operário/coordenador. Qualquer um dos agentes pode executar as operações do passo substituindo o operário em caso de falha. Tais operações são confirmadas com a participação de todos os agentes, como no protocolo em [Ple02].

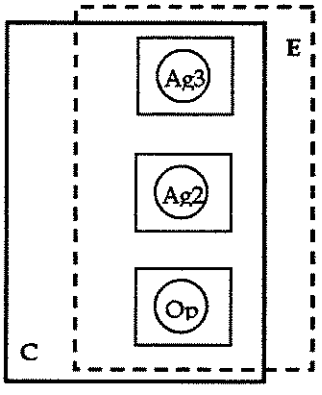


Figura 4.4 - Esquema MMJ

MUD: Este esquema se adapta a protocolos que realizam a coordenação depois do passo, aqui a execução pode desencadear paralelamente em diferentes agências, porém apenas o primeiro agente que alcançar a agência de destino, terá sua família (execução) confirmada, ex [MPT00].

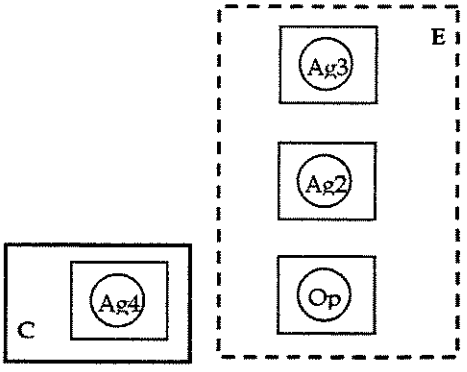


Figura 4.5 - Esquema MUD

4.2 Utilizando exceções para proteger o agente

Exceções podem ser uma das formas mais simples de coordenar a replicação em um sistema de agentes [TM00]. Para cada agente é definido um guardião. Ele é utilizado quando ocorre uma exceção que o agente não é capaz de tratar ou quando a agência detecta a parada do agente. O guardião pode ser visto como uma réplica solitária, embora não possua o mesmo código nem estado do agente que realiza a tarefa. Quando for gerada uma exceção a qual o agente não sabe tratar, ele será transportado para junto do guardião que decidirá qual procedimento seguir, por meio de um vetor que indica, para cada nó visitado pelo agente, se a condição é normal ou a exceção gerada. A programação do guardião é responsabilidade do desenvolvedor da aplicação que poderá encerrar a atividade do agente, reiniciar a tarefa, etc, de acordo com a semântica da aplicação.

Exemplo

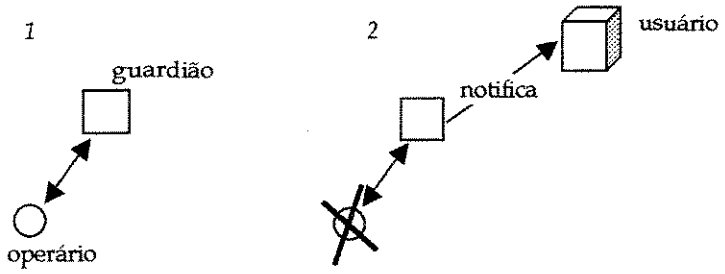


Figura 4.6 - O operário falha e seu estado é enviado ao usuário.

Um problema do modelo apresentado é a ocorrência de partições da rede que venham a isolar o agente de seu guardião, paralisando o tratamento de exceções distribuídas. Portanto, esse mecanismo pressupõe que o sistema se comporte como descrito no modelo falha e pára.

4.3 Definindo a tarefa do agente como uma transação

Em [VB98], [VKM97] foi apresentado um protocolo baseado no serviço de transações CORBA. Em cada agência visitada pelo operário, é criado um agente estacionário que atuará como interface entre o recurso e o gerenciador de transações. O recurso não necessita apresentar semântica transacional; os agentes monitoram recursos em uma fase de preparação e executam a transação em uma fase posterior. No caso de todos os recursos serem cooperativos e assinalarem que entraram no estado de *prepared to commit*, é possível realizar a transação segundo o modelo aninhado fechado.

Como última operação de sua tarefa, o operário "montará" (registrará) a transação junto ao serviço de transações CORBA, sendo chamado nesse momento de *originator*.

1 O operário, agindo como um cliente transacional, inicia a transação invocando o serviço de transações CORBA para criar um novo contexto.

2 O operário interage com os agentes estacionários, para registrá-los no contexto da transação distribuída.

3 Os agentes estacionários, que desempenham o papel definido como *recoverable object* na especificação CORBA, atuam como interface entre o gerenciador de transação CORBA e os recursos. Os agentes estacionários são responsáveis por sinalizar ao operário a impossibilidade do recurso de participar da transação. Essa sinalização pode ser realizada logo após o operário requisitar a operação, podendo levá-lo a abortar sua tarefa e, assim, evitar retrocessos complexos.

4 Os recursos são registrados pelos agentes estacionários no serviço transacional CORBA (OTS) [TS01].

5 Finalmente os recursos participam do *commit* de duas fases.

Em [AS99] a tarefa dos agentes é definida em um modelo que integrou transações aninhadas e compensação (veja seções 3.5.1 e 3.5.2). O desenho a seguir ilustra uma tarefa para a qual foram definidos todos os tipos de transações do modelo:

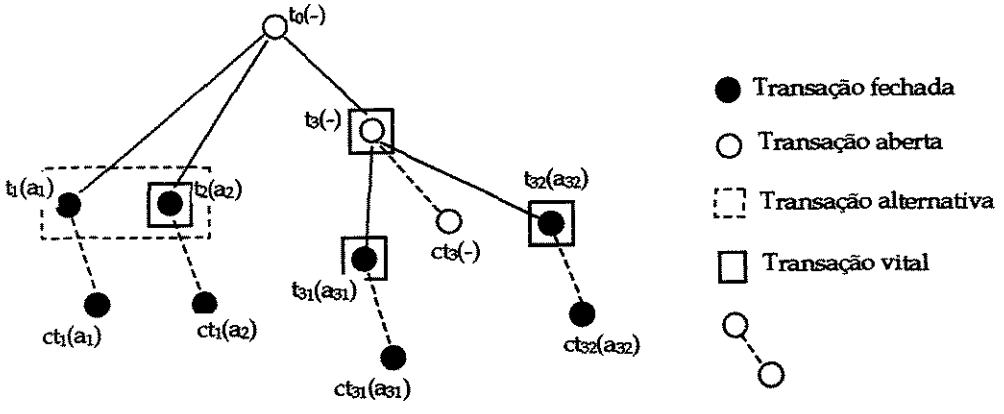


Figura 4.7 - Árvore de transações com exemplos: abertas, fechadas, vitais, compensatórias, alternativas

No desenho, t_1 e t_2 são alternativas associadas às compensatórias ct_1 e ct_2. Ambas são *flat* e compõem uma transação vital, aquela que não sendo completada implicará no fracasso da transação global. t_3 é uma transação aberta.

O grupo de agentes é organizado de forma que cada membro resida em uma agência onde foi definida uma transação alternativa. Caso o operário falhe, o novo operário pode iniciar a transação alternativa sem a necessidade de reposicionamento. Pré-posicionar, portanto, os membros em agências onde estão definidas transações alternativas acelera a recuperação de uma falha.

Na ocorrência de falha, tentar-se-á executar a recuperação por avanço; não sendo possível, prossegue-se com a recuperação por retrocesso. É possível que ocorra recuperação por retrocesso parcial, executando-se algumas transações compensatórias em conjunto com transações alternativas.

Um agente pode mover para a próxima etapa após receber a resposta de todas as ações aninhadas.

A escolha pelo modelo de transações aninhadas abertas foi justificada a partir da análise de um exemplo caracterizado por ações de longa duração. Como discutido na seção 1.4.5, há aplicações que exigem um modelo diferente do aninhado aberto como, por exemplo, a atualização de *software*. Essas aplicações podem ser executadas em uma intranet, apresentando requisitos diferentes daqueles analisadas em [AS99].

4.4 Um protocolo de tolerância a falhas baseado em *broadcast*

Em [JMS+98] foi apresentado um protocolo em que o conjunto de agentes realiza cada etapa da tarefa como uma ação atômica. Neste modelo, o deslocamento também faz parte da ação (migração atômica) que é finalizada por uma instrução de migração ou clonagem.

Para cada ação A, é associada uma ação de recuperação $\tilde{A}$. Durante a execução:

- A é executada no máximo uma vez, com ou sem falha;

- Se A falhar, Ā é executada pelo menos uma vez e uma vez sem falha;
- Ā será executada se e somente se A falhar.

O protocolo inicia-se com um único agente que a cada migração deixa uma réplica, denominada *rear guards*, na agência de partida, até que o número de réplicas atinja um valor predefinido.

Inicialmente a ação A é executada pelo operário. Caso não seja possível completar a ação A, o operário tentará executar Ā repetidamente, até que seja obtido sucesso. No caso de destruição do operário, a réplica imediatamente anterior passará a executar Ā.

Apesar de elegante a definição da atomicidade no par <A, Ā>, é possível obter o mesmo efeito em outros sistemas que não definiram a execução do agente desta maneira, pela programação do itinerário baseado no tratamento de exceções. Ā pode ser entendido como um tratador de exceção, sendo apenas necessária a notificação do evento de falha para dispará-lo.

A probabilidade de paralisação da tarefa é maior nas etapas iniciais. A replicação não é aproveitada para introduzir paralelismo ou acelerar a tarefa global.

O itinerário começa a ser executado por um único agente. Ao início de cada passo, é criado um novo agente para executá-lo (operário) enquanto os mais antigos o monitoram através das mensagens `I am alive` que o operário envia periodicamente.

O itinerário, juntamente com parte do estado do agente, é representado em uma estrutura denominada *briefcase*, composta por conjunto de pares <nome, valor>, os *folders*. A *briefcase* é organizada em forma de lista, cada elemento contendo informações de um determinado passo. Conforme a execução progride, os *folders* correspondentes aos passos já executados são excluídos. Uma *briefcase* é composta por:

HOST: lista das agências que compõem o itinerário.
CODE: lista de ações.
RECOVERY: lista de ações de recuperação.
VERSION: versão da ação atual.
NUM_GUARDS: número ideal de réplicas (*rear guards*).
RALLY_POINT: lista de agências para a qual o agente deve migrar em condições de erro extremas.
RECOVERY_HOSTS: lista de agências onde são executadas as ações de recuperação.
FAILURE_STATUS: contém informações sobre exceções geradas e mensagens de falha.

Figura 4.8 - Estrutura de dados utilizada pelo protocolo *NAP* (*folders*)

No momento em que é iniciado um novo passo, a *briefcase* é distribuída a cada um dos agentes que participarão do referido passo. Isso é feito de acordo com o *NAP* (*Norwergian Army Protocol*): a *briefcase* é entregue do agente mais recente para o mais antigo. Cada agente entrega a *briefcase* para seu antecessor até que o último a tenha recebido, neste momento, o último agente iniciará uma sucessão de mensagens de confirmação, que, por fim, chegará ao operário (agente mais recente). Dois agentes consecutivos da cadeia realizam monitoração recíproca; se um dos agentes falhar a cadeia é refeita entre sucessor e antecessor do agente falho; de fato, um agente é responsável por todos seus antecessores. Quando o(s) agente(s) mais antigo(s) falha(m), seu sucessor passa a ocupar a última posição; semelhantemente, quando a réplica `f+1` recebe a *briefcase*, ela passa a ocupar o fim da fila e as réplicas subseqüentes podem ser eliminadas. Em suma, o *NAP* realiza

um *"broadcast"* de informações relacionadas com o passo (*briefcase*) para `f+1` réplicas.

Existem três resultados possíveis para o *broadcast* de uma *briefcase* `b` ao fim do passo *i*:

Nenhum agente recebe `b`: o operário falhou e uma réplica executará $\bar{A}_i$.

O agente a_{i+1} recebe `b`: todos os agentes não falhos também receberam `b` e o operário a_{i+1} dará prosseguimento à execução.

Alguns agentes receberam `b`, mas o agente a_{i+1} falhou: alguma réplica irá executar a ação $\bar{A}_{i+1}$.

Este trabalho assume o modelo falha e pára. Caso o grupo de agentes seja dividido por uma falha de comunicação, teremos o operário executando `A` e uma réplica (na outra partição) executando $\bar{A}$, violando a propriedade da unicidade da execução.

Apesar de ser possível excluir um número qualquer de réplicas do grupo, apenas uma nova réplica pode ser incluída a cada passo (apenas um novo operário pode ser criado a cada passo). Desse modo, o *NAP* progride apenas em replicação linear.

Este trabalho é classificado como MMJ, apesar de não haver realizado consenso: a *briefcase* deve ser distribuída a todos os agentes do grupo.

Embora seja o operário o único a executar a ação normal `A`, todas as réplicas podem executar $\bar{A}$; disso decorre a classificação de execução múltipla. O último agente a receber a mensagem de confirmação é o operário, daí decorre o "J" do MMJ.

Está prevista a operação de *checkpoint* em que é possível persistir o estado do agente para ser utilizado por ações de recuperação.

Exemplo

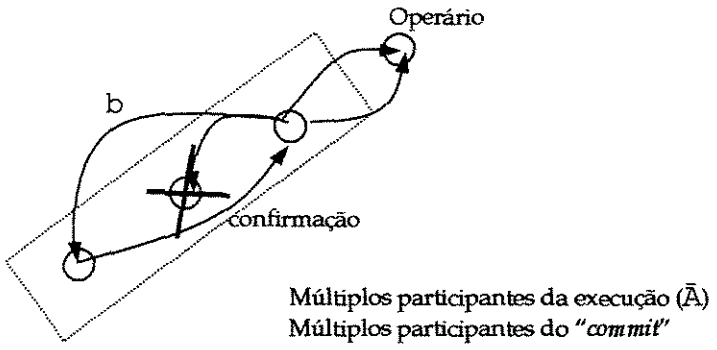


Figura 4.9 - execução do NAP - Norwergian Army Protocol (MMJ). A briefcase b é enviada a cada um dos membros do grupo do passo que está sendo iniciado. Ocorre uma falha e a cadeia é refeita. Por fim, a última mensagem de confirmação chega ao operário que, ciente da existência de f+1 agentes em seu grupo, dá início às operações do passo

4.5 Um protocolo de tolerância a falhas baseado no *commit* de duas fases

Para garantir as propriedades de continuidade e unicidade da execução, foi apresentado em [RS98] uma abordagem baseada em três protocolos: monitoração, *commit* do passo (também chamado de votação ou término do passo) e eleição (também chamado de seleção).

O protocolo de monitoração exige que o operário envie periodicamente mensagens `I am alive` para todas as réplicas.

Quando é detectada a falha do operário, uma das réplicas é eleita operário substituto.

Em caso de partições na rede é possível que dois operários coexistam, porém o protocolo de *commit* garante que apenas um conseguirá completar o passo. Cada agência dispõe de um fila que implementa operações transacionais, para receber agentes. Imagine um grupo de agentes executando um passo *i*. Suponha que apenas um agente execute as operações. No momento em que o operário a_i conclui

a última operação, ele envia uma cópia do novo agente a_{i+1} para as agências que participarão do passo *i+1*. Para poder realizar o *commit* da transmissão dos agentes, o operário (que passa a se chamar *Orquestrator*) desempenhará a função de coordenador, necessitando da aprovação da maioria dos agentes de seu grupo.

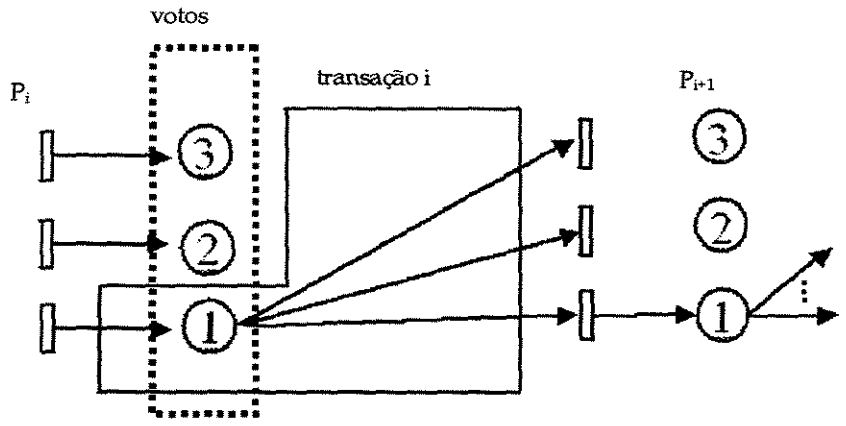


Figura 4.10 - O operário "1" (passo p_i) recolhe votos dos membros para realizar o *commit* do envio dos agentes ao passo p_{i+1}.

Portanto, o modelo de falhas deste trabalho admite partições da rede. Em caso de falha, primeiro é eleito um novo operário para, no momento da conclusão do passo, serem eliminados operários redundantes, garantindo a unicidade da execução.

Dessa forma o problema da eleição e do *commit* das atividades do agente estão vinculados, o que representa uma fraqueza pela possibilidade do sistema ficar bloqueado caso o coordenador venha a falhar. Ainda que o protocolo de *commit* seja substituído pelo *commit* de três fases, é necessário que as réplicas se comprometam com o coordenador para garantir que apenas este último conclua o protocolo (e, conseqüentemente, finalize o passo).

Neste protocolo, a probabilidade de bloqueio da tarefa é igual em todas as etapas devido à quantidade de réplicas ser constante. Uma vez que as réplicas residem em agências adjacentes, não necessariamente já visitadas pelo agente

principal, é possível aproveitar a redundância para introduzir paralelismo. Também é possível que os agentes progridam segundo a replicação linear, embora a cada novo passo, seja enviada uma cópia do novo operário a todas as agências participantes.

Este trabalho é classificado como MUJ.

Apesar de não serem encontrados no texto informações sobre a utilização de *checkpoint*, é possível persistir o estado do agente e fazer o novo operário recuperar esse estado.

Exemplo

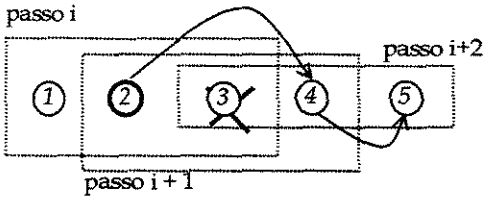


Figura 4.11 - Falha durante a execução do protocolo apresentado em [RS98] (utilizando-se replicação linear)

Passo *i*:

O operário `(3)` falha. As demais réplicas detectam a falha pela ausência das mensagens `I am alive` e iniciam o protocolo de eleição. Uma réplica `(2)` é eleita novo operário e executa as ações do passo *i* e o finaliza.

Passo *i+1*:

Como as agências que participam do passo foram predefinidas e `(3)` continua falha, somente `(2)` e `(4)` participam do passo. Note que há reutilização de agências. Apesar disso, o protocolo exige que o novo agente seja enviado para todas as agências do passo. Para que o sistema se beneficie da reutilização, é

necessário implementar algum tipo de otimização que exija a transferência apenas do estado do novo agente, reaproveitando o código que já está na agência.

Passo *i*+2:

O sistema progride sem problemas. Como a agência `(3)` continua falha, participam do passo somente `(4)` e `(5)`.

4.6 Um protocolo de tolerância a falhas baseado no *commit* de três fases

O trabalho [AS99] baseia-se naquele apresentado na seção anterior, contornando a questão do bloqueio do sistema devido a falhas do coordenador. Para isso é utilizado o registro persistente (*log*) do resultado do protocolo de seleção e do estado corrente do passo. Para um operário realizar a finalização do passo, além da maioria dos votos, é necessário certificar-se de que ele é o único operário em execução em sua partição e que o passo ainda não foi concluído.

Imagine que devido ao surgimento de uma partição, um segundo operário seja eleito e passe a executar o mesmo passo. Como foi dito, a eleição de um novo operário é registrada em mídia segura.

1 Se a partição perdurar, somente um operário conseguirá finalizar o passo devido à exigência de $\left\lceil \frac{n+1}{2} \right\rceil$ votos (isto é registrado). Se a partição desaparecer, o outro operário perceberá que o passo já acabou e abortará suas atividades.

2 Se a partição desaparecer antes da finalização, o operário mais antigo descobrirá, a partir do *log*, que outro operário foi eleito e abortará.

Figura 4.12 - Estratégia para garantir a unidacidade da execução por meio do protocolo de *commit*

Com a persistência das informações relacionadas à finalização do passo, é possível desvincular o problema da eleição do operário do *commit* do passo. Substituindo-se o protocolo de *commit* de duas fases pelo *commit* de três fases, previne-se o bloqueio do sistema pela falha do coordenador, sem prejuízo da propriedade de unicidade da execução. O preço da melhoria está na necessidade de se utilizar um repositório distribuído acessível a partir de qualquer agência, denominado DCD (*Distributed Context Database*), e que também pode ser utilizado para registrar *checkpoints*.

Quando um agente é criado ou muda de passo, seu *checkpoint* é registrado no DCD. O agente realiza operações definidas no passo e, ao final, participa de um protocolo de *commit* de três fases. O operário é constantemente monitorado, e, caso fique indisponível por um longo período, uma nova réplica primária será eleita. A computação será reiniciada a partir do último *checkpoint* registrado no DCD.

Uma agência pode se encontrar em um dos seguintes estados:

- *UNKNOWN*: não há informações a respeito da agência (está falha);
- *QUEUED*: contém um agente recém-chegado que está pronto para iniciar a execução;
- *ACTIVE*: contém um operário que está executando normalmente;
- *MONITORING*: contém uma réplica que monitora o operário;
- *SHORT_TERM_FAILURE*: a agência acaba de ser reiniciada e está avaliando a extensão da falha;
- *PROCESSING_LONG_FAILURE*: a agência detectou a eleição de um operário substituto enquanto encontrava-se parada e desfará as operações do agente operário que nela executava (*rollback*);
- *TERMINATING*: contém um operário que está realizando o *commit* do passo;

Figura 4.13 – Estados (possíveis) de uma agência

Exemplo

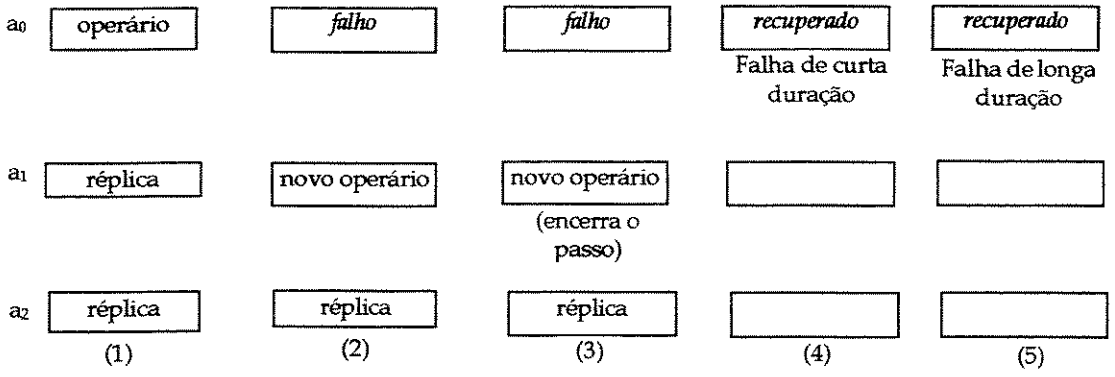


Figura 4.14 – Recuperação de uma falha longa através da eleição de um novo agente operário

1 No início de uma etapa, o operário está na agência com maior prioridade (neste exemplo, aquela com o menor índice – a_0).

2 Ocorre uma falha que provoca a parada da agência a_0. O mecanismo de recuperação é automático, entrando em operação assim que uma falha é detectada. Uma nova réplica é eleita (na agência a_1) iniciando a leitura do *checkpoint* gravado no DCD. É certo que o novo líder sempre lerá o *checkpoint* mais atualizado.

3 O passo é finalizado pelo agente de a_1.

4 A agência a_0 se recupera e avalia a extensão da falha (falha de curta duração).

5 A agência percebe que o passo já foi terminado; são apagadas informações registradas no DCD após o *checkpoint* a partir do qual o operário começou a executar (falha de longa duração).

Este trabalho é classificado como MMD. As réplicas podem vir a executar instruções do passo, se eleitas operário. O segundo "M" do MMD deve-se à participação das réplicas no protocolo *commit* de três fases. O "D" decorre do

resultado ser gerenciado por um processo externo ao sistema de agentes. O *checkpoint* faz parte do protocolo de finalização do passo, sendo realizado no momento em que é transmitido o agente que executará o passo seguinte.

Pode ser utilizada a replicação disjunta ou linear, porém o novo agente é transmitido a todas as agências, sendo necessário programar algum tipo de otimização para tirar vantagem da sobreposição de agências de passos consecutivos, evitando que o estado completo seja transmitido a cada uma das agências.

Este trabalho foi elaborado no modelo com falha de partição e possui três protocolos:

- **monitoração**: consiste no envio periódico de mensagens `I am alive` pelo operário a cada uma das réplicas;

- **eleição**: diferencia-se pelo fato da identificação do novo operário ser persistida;

- **finalização**: é realizada através da execução do protocolo de *commit* de três fases.

4.7 Um protocolo de tolerância a falhas baseado no consenso distribuído

Para garantir a unicidade da execução o passo do agente foi definido em [PS00], [PS01], [Ple02] como um problema de consenso. Os membros de um grupo de agentes devem concordar, ao final de cada passo, em:

- qual o agente que atuou como operário (e conseqüentemente seu estado final);
- quais agentes participarão do passo seguinte;
- qual agente será o operário do passo seguinte.

Uma solução deve obedecer a três propriedades:

- (Consenso) Dados dois agentes não falhos, estes apresentam o mesmo resultado;
- (Validade uniforme) Se um agente decidir por um valor S_i para o passo *i*, então S_i foi proposto por um operário que executou o passo *i*;
- (Integridade uniforme) Cada agente participante do passo *i* decide no máximo uma vez.

Para o sistema não bloquear, a propriedade de continuidade da execução exige que os agentes cheguem, em algum momento, ao consenso, o trabalho propõe a utilização do protocolo DIV-consensus (*Defered Initial Value*) para resolver o problema.

Neste trabalho o agente que executará um novo passo, é enviado a todas as agências participantes. Isso diminui a necessidade de realizar *checkpoints*, pois cada réplica possui o estado do agente operário no momento em que este iniciou o passo. É possível persistir o estado do operário em momentos intermediários do passo (até em um repositório distribuído acessível a todas as réplicas). É possível também utilizar a replicação disjunta ou linear.

Este protocolo admite partição e pode ser utilizado em conjunto com a replicação linear ou replicação disjunta.

4.8 Um protocolo de tolerância a falhas baseado no *flooding* de agentes

Em [MPT00], é apresentado um protocolo de coordenação de agentes que realiza o controle ao final da execução. Este protocolo utiliza replicação linear; cada agente monitora um ou mais sucessores através das mensagens `I am alive` que devem ser enviadas regularmente. A freqüência das mensagens é inversa à distância entre os agentes.

Quando é detectada a falha de um agente, seu antecessor mais próximo recria uma nova réplica. Caso o agente tenha sido destruído devido à falha de uma agência, e esta continue parada, a nova réplica tentará executar as operações em outra agência que ofereça serviços similares, percorrendo assim um itinerário alternativo. Devido à impossibilidade de um agente diferenciar a parada de seu sucessor do isolamento deste por problemas de comunicação, é possível que dois ou mais agentes realizem as mesmas operações. Entretanto somente o agente que chegar primeiro terá seu itinerário confirmado, garantindo, assim, a propriedade da unicidade da execução. As réplicas que chegarem depois deverão abortar operações (repetidas). Como não é possível uma réplica saber se as operações de sua família serão confirmadas, é necessário manter todas as trancas até atingir o destino (modelo de transações fechadas), o que implica na redução da concorrência do sistema.

4.9 Uma comparação entre os trabalhos relacionados

Segue uma tabela listando os principais trabalhos nos pontos abordados anteriormente:

	Exceções [TM00]	*NAP* [JMS+98]	[RS98]	[AS99]	NetPebbles [MPT00]	[Ple02]	Protocolo Capítulo 5
Protocolo	Não tem	*NAP* (broadcast)	*Commit* 2 Fases	*Commit* 3 Fases	Broadcast	DIV-Consesus	Lista (não limitado no tempo)
Replicação	Somente Linear (réplica única)	Somente Linear	Linear/ Disjunta	Linear/ Disjunta	Somente Linear	Linear/ Disjunta	Linear/ Disjunta
Modelo de falha	Falha e pára	Falha e pára	Partição	Partição	Partição	Partição	Partição
Checkpoint	Não	Sim	Sim	Sim	Sim	Sim	Sim
Classificação	UUJ	MMJ	MUJ	MMD	MUD	MMJ	MMJ
Commit	Depois do passo	Depois do passo	Depois do passo	Depois do passo	**No destino**	Depois do passo	Depois do passo

Tabela 4.1 - Comparação entre sete protocolos de coordenação de agentes

Capítulo 5 - Um Protocolo de Tolerância a Falhas para Sistemas de Agentes Móveis

Este capítulo aborda a questão de tornar um sistema de agentes móveis disponível mesmo na presença de falhas (veja o modelo de falhas na seção seguinte). Como nos paradigmas tradicionais (ex: cliente-servidor) é utilizado replicação para que a parada de um agente não implique no colapso do sistema de agentes. Entretanto o caráter ativo dos agentes abre a possibilidade da reexecução de operações quando um ou mais agentes suspeitam incorretamente que outro membro falhou. É necessário coordenar as réplicas assegurando que cada operação seja executada uma única vez.

O problema da garantida da unicidade da execução explicado no parágrafo anterior é abstraído como o problema de organizar a execução da tarefa dos agentes em uma sucessão de listas de agentes, onde cada lista define um agente operário e um conjunto de réplicas. Se cada passo é associado a uma única lista, cada operação será executada uma única vez. Resta desenvolver um protocolo para criar listas obedecendo a restrição onde uma lista A só poderá originar uma única lista A'. Este protocolo coordenará o grupo de agentes, definindo (através de lista)

quais novos agentes serão criados/destruídos e para quais agentes será concedido/revogado o direto de voto (de aprovar uma nova lista).

5.1 Modelo de falhas

O protocolo apresentado nesta seção pressupõe que os agentes não se comportam de maneira maliciosa. É possível que mensagens sejam perdidas ou sejam recebidas fora de ordem, isto é, uma mensagem `A` enviada antes de uma mensagem `B`, `pode` atingir o destinatário depois de `B`. Toda mensagem `A` recebida, porém, é exatamente igual à mensagem `A` enviada, isto é, as mensagens são incorruptíveis. Finalmente um ou mais agentes podem ficar isolados do restante do grupo devido ao surgimento de partições, que podem ser permanentes ou temporárias.

5.2 Introdução

O objetivo é utilizar replicação para tolerar falhas evitando, contudo, que dois ou mais agentes reexecutem uma mesma tarefa pela impossibilidade de detectar falhas com precisão, ex: um operário incomunicável está falho ou apenas isolado em uma partição? É utilizado o conceito de grupo definido por uma lista onde também é indicado o membro que atua como operário. Quando é necessário substituir o operário, seja porque falhou ou porque terminou de executar as operações do passo (é utilizado um operário por passo), executa-se o protocolo para que o grupo se auto-redefina por meio da aprovação de uma nova lista.

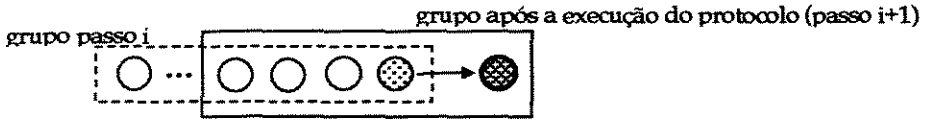


Figura 5.1 - Reconfiguração do grupo de agentes a medida que a avança a execução da tarefa

Qualquer agente pode propor aos demais membros uma nova lista, mas essa só será criada com aprovação da maioria. O número de integrantes do grupo permanece constante (na seção 5.8 veremos que é possível flexibilizar esta exigência), isto é, para que um novo agente seja criado, outro será excluído e eliminado. Na seção 5.7 mostraremos que as duas propriedades são suficientes para garantir a propriedade da unicidade da execução: a necessidade da maioria aprovar a nova lista implica na possibilidade de criar uma única lista `L'` a partir de uma lista anterior `L`, enquanto o número de integrantes do grupo se mantém constante, permitindo definir a maioria também como uma constante evitando que agentes perdidos (que não participaram de algumas execuções do protocolo por estarem isolados em outra partição) possam se combinar para formar uma lista.

Garantido que qualquer lista `L` poderá ser sucedida por uma única lista `L'` e sabendo-se que uma lista define apenas um único operário conclui-se que cada tarefa será executada uma única vez.

O grupo de agentes é definido por uma lista de n agentes entre os quais um único executa as operações; os demais se limitam a monitorá-lo e proceder a sua recuperação, em caso de falha; n é o número de cópias do sistema, definido pelo usuário, isto é, o grau de replicação. É possível existir um número inferior a n agentes, no sistema, caso falhas destruam *alguns* desses. Cada lista possui a indicação de qual agente é o operário e um número inteiro positivo crescente que define sua versão.

No momento que é iniciado um novo passo ou quando é necessário substituir um operário falho, é criada uma nova lista, portanto um passo está relacionado a uma ou mais listas. Na seção 2.1.1 foi explicado que migração envolve destruir um processo `p1` em uma máquina `m1`, transmitir seu estado para outra máquina `m2` onde será criado um processo idêntico (utilizando o mesmo código e no mesmo estado que `p1` se encontrava no momento imediatamente antes da migração); de fato a migração é uma redefinição do grupo. Para garantir que a migração foi

completada com sucesso, alguns trabalhos sugeriam a utilização de um "protocolo de migração atômico" [JMS+98]. Este protocolo tinha como função garantir que o agente realmente conseguiu atingir seu novo destino. O protocolo aqui descrito desempenha exatamente a função do protocolo de migração atômico quando utilizado no momento da migração, isto é, quando um novo passo é iniciado. O número de agentes na lista é constante, portanto, o novo operário estará substituindo o antigo.

Cada agente armazena consigo uma lista cuja versão pode diferir em relação à lista utilizada por outro agente do sistema.

Lista válida é aquela que foi aprovada, para tanto foi aceita por, no mínimo, $\left\lceil \frac{n+1}{2} \right\rceil$ agentes da lista válida anterior, portanto:

- para uma nova lista ser criada, é necessário um quorum de no mínimo $\left\lceil \frac{n+1}{2} \right\rceil$ agentes;

- para que um agente possa participar desse quorum, é preciso que pertença à última lista válida criada no sistema.

Os agentes que não fazem parte da última lista válida, não podem se combinar, sendo chamados de inativos.

O problema da monitoração de falhas foge ao âmbito deste trabalho, sendo possível aproveitar técnicas descritas na literatura como, por exemplo, o envio periódico de mensagens *I am alive* pelo operário para cada uma das cópias.

Quando um agente suspeitar que o operário falhou (corretamente ou não) ele se tornará coordenador ao convidar os outros agentes de sua lista atual, que desempenharão papel de participantes. É possível ocorrer conflito quando dois agentes se tornarem coordenador simultaneamente, de forma análoga a uma "colisão" em redes Ethernet. Após uma tentativa fracassada de criar a lista (o conflito impede o coordenador de obter quorum necessário para aprovar a lista -

veja exemplo 3 da seção 5.6), os agentes deverão aguardar um tempo aleatório para realizar uma nova tentativa. Em suma, o coordenador da ação de recuperação é escolhido aleatoriamente. Na ausência de falhas o coordenador será o operário.

Um passo é encerrado quando surgir no sistema uma nova lista válida, isto é, quando o coordenador conseguir distribuir a nova lista para $\left\lceil \frac{n+1}{2} \right\rceil$ agentes. Cada agente que recebe a nova lista deixa de usar a antiga; desta forma quando o $\left\lceil \frac{n+1}{2} \right\rceil$-ésimo agente passar a usar a nova lista, não haverá mais $\left\lceil \frac{n+1}{2} \right\rceil$ agentes usando a antiga, isto é, a mudança de lista válida é uma operação atômica.

5.3 Execução sem falhas

Vamos iniciar a apresentação do protocolo descrevendo sua operação na ausência de falhas. Na próxima seção será discutida a execução do protocolo na presença de falhas. Na seção 5.5 será apresentado o algoritmo e, em seguida, exemplos de sua execução; na seção 5.7 será discutida a correção do algoritmo, em seguida será apresentada uma extensão do algoritmo para listas de tamanhos variável; na seção 5.9 será discutida a gerência de estado do agente operário e finalmente na seção 5.10 será analisada a complexidade do protocolo.

A cada novo passo o grupo de agentes é redefinido: novos agentes são criados (entre eles um novo operário), enquanto agentes antigos são eliminados.

A substituição do agente mais antigo por um novo operário (replicação linear) apresenta o menor custo, entretanto é possível substituir um ou mais agentes (replicação vertical).

Para criar uma nova lista, o coordenador (operário) deve pertencer à última lista válida criada no sistema e propor uma nova, por meio do envio de um convite para os agentes de sua lista. O coordenador necessita do consentimento de pelo

menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes; obtendo sucesso, ele envia a nova lista a cada um dos agentes de sua lista antiga. Os novos agentes são criados após o coordenador obter a confirmação de que pelo menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes receberam a nova lista. Ao receber uma lista da qual não faz parte, um agente morre.

5.4 Execução com falhas

Falhas podem ocorrer durante a execução normal e até mesmo a recuperação. Se o operário falhar, deve ser substituído, o que exige a criação de uma nova lista; qualquer agente pode iniciar a substituição do agente falho, sendo denominado, a partir deste momento até o encerramento do protocolo, coordenador. A criação da lista para recuperação do grupo segue as mesmas etapas descritas na seção anterior.

É possível que dois ou mais agentes iniciem a execução do protocolo simultaneamente, porém somente um (ou nenhum) conseguirá obter $\left\lceil \frac{n+1}{2} \right\rceil$ votos da última lista válida.

A falha de um membro (não operário) não resultará em uma ação de recuperação porque esta não interromperá a execução da tarefa, apenas poderá bloquear o sistema no momento do passo, caso não seja possível formar um quorum com pelo menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes. Talvez a falha seja temporária, porém caso o agente não se recupere, será substituído em uma lista futura.

Se o protocolo não for completado com sucesso, o coordenador deixará de desempenhar esse papel, podendo voltar a propor uma nova lista (caso o sistema não se recupere), em um momento posterior. Um agente deverá aguardar um tempo aleatório entre duas tentativas, que pode ser determinado da seguinte

maneira: seja r um número randômico entre zero e t, onde t é o número de tentativas fracassadas; o tempo de espera será igual a 2^t.

5.5 Protocolo

Primeiro é verificado se os agentes reunidos podem se combinar para criar uma nova lista, isto é, se fazem parte da última lista válida criada no sistema. Isso é determinado com a ajuda do número de versão da lista utilizada por cada agente. Recordando, a lista válida é aquela utilizada por pelo menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes no sistema e não possui necessariamente o maior número de versão.

Suponha o seguinte histórico de listas válidas: S → S' → S" e um coordenador C definido exclusivamente em S. Isto é, o grupo de agente já prosseguiu na execução da tarefa, sendo criadas até o momento 3 listas: S, S', S". Neste ponto um agente antigo C definido na lista S, mas não em S' ou S" tenta criar uma nova lista (C não sabe da existência de S', S"). Para que S' fosse criada, pelo menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes definidos em S receberam uma cópia de S'(cada agente só envia a confirmação após receber a cópia da nova lista); quando C contatar os agentes de S (C imagina que o sistema não evoluir além de S, e tenta repetir o passo correspondente a S') em busca de um quorum de $\left\lceil \frac{n+1}{2} \right\rceil$ agentes, pelo menos um utilizará S' e o informará de sua existência enviando-lhe uma cópia. C conhecerá os agentes de S' que não estão em S, e prosseguirá recursivamente até encontrar a última lista válida, neste caso S". Nesse momento C, saberá:

1. Os agentes que pertencem ou não a última lista válida.
2. Se ele próprio pertence a ela.
3. Se C obteve o voto de $\left\lceil \frac{n+1}{2} \right\rceil$ agentes da última lista válida.

Segue, o protocolo:

Coordenador:

Fase I: Se a falha do operário for detectada ou for o momento de dar um passo, envia convite (ID do coordenador, número de versão) para os agentes que conhece.

Fase II: Aguarda até que todas as respostas sejam recebidas ou TIMEOUT.

{ Se ocorrer TIMEOUT, o algoritmo retorna à fase I }

Se uma lista mais recente for recebida: *{procurando a última lista válida}*

Se forem descobertos novos agentes envia convite para cada um; reinicia o temporizador; retorna ao início da *fase II*.

Do contrário envia aborta, espera e recomeça.

Se o coordenador ainda não utiliza a última lista válida: assuma-a. O protocolo termina.

Se forem recebidos $\left\lceil \frac{n+1}{2} \right\rceil$ "sim" de agentes da última lista válida: envia a nova lista para todos os agentes da última lista válida (com um número de versão maior).

Do contrário envia aborta, espera e recomeça.

Fase III: Aguarda até que todas as confirmações sejam recebidas ou TIMEOUT.

Se $\left\lceil \frac{n+1}{2} \right\rceil$ "confirmado" forem recebidos, cria o(s) novo(s) agente(s) com ID igual(is) àquele(s) definido(s) na nova lista.

Participante:

Fase I: Se um "convite" for recebido:

Se o número de versão contido no convite for maior ou igual ao número de versão da lista do participante:

Se o participante não estiver comprometido: responda "sim", comprometendo-se.

Do contrário responda: "não".

Do contrário responda "não", junto com a lista do participante.

Fase II: Aguarda nova lista ou TIMEOUT.

Se a nova lista for recebida passe a utilizá-la. Responda "confirmado".

O agente está liberado do compromisso.

A última lista válida não possui, necessariamente, o maior número de versão existente no sistema; uma lista inválida pode ter um número maior. Uma lista inválida `I` corresponde a uma tentativa fracassada na qual `I` foi distribuída para menos de $\left\lceil \frac{n+1}{2} \right\rceil$ agentes. `C` saberá que encontrou a última lista válida `V` criada no sistema, quando contatar pelo menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes definidos em `V` que estão utilizando `V`. Como foi dito, `V` não pode ser inválida porque foi distribuída a $\left\lceil \frac{n+1}{2} \right\rceil$ agentes; não pode haver uma lista válida `V'` mais recente que `V`, porque pelo menos um dos agentes contatados por `C` já estaria utilizando `V'`.

Se o coordenador não estiver utilizando a última lista válida sua tentativa será abortada, mesmo que faça parte da lista.

À medida que uma nova lista L é distribuída a agentes que dela não fazem parte, eles são eliminados, porém os novos agentes definidos em L só serão criados

quando $\left\lceil \frac{n+1}{2} \right\rceil$ mensagens "confirmado" forem recebidas pelo coordenador. Quanto mais agentes forem substituídos, maior é o perigo de o sistema bloquear se L não for aprovada. Apesar de ser possível utilizar o presente protocolo com replicação disjunta é aconselhável substituir um número pequeno (de preferência um único) de agentes a cada passo porque caso o coordenador venha a falhar o sistema poderá ficar bloqueado por falta de quorum, uma vez que os novos agentes são criados próximos ao encerramento do protocolo.

5.6 Exemplos

Exemplo 1 - **Falha temporária**

Surge uma partição e um dos agentes isolados detecta falha do operário, iniciando a criação de uma nova lista; entretanto sua partição não contém $\left\lceil \frac{n+1}{2} \right\rceil$ agentes, não é possível prosseguir; os agentes isolados continuarão tentando até que a falha desapareça. O retorno do operário (desaparecimento da partição) resulta no cancelamento da ação de recuperação.

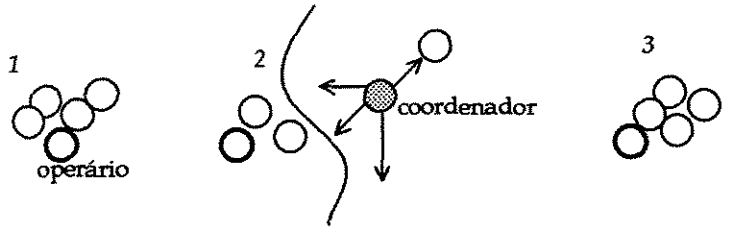


Figura 5.2 – Exemplo de falha temporária.

Exemplo 2 - Falha de longa duração

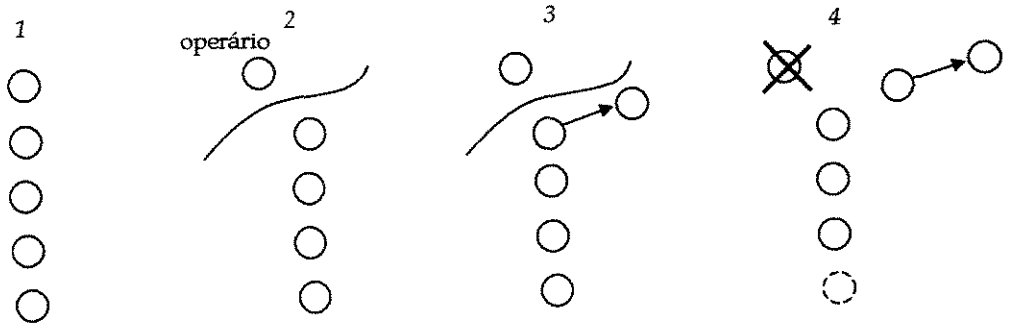


Figura 5.3 - Falha de longa duração.

1. Surge uma partição e o operário é isolado.
2. O operário não conseguirá dar o passo, está bloqueado.
3. Na outra partição, os agentes criarão uma nova lista com um operário substituto.
4. Quando o antigo operário perceber que não faz parte da última lista válida, morre.

Exemplo 3 - Concorrência entre coordenadores

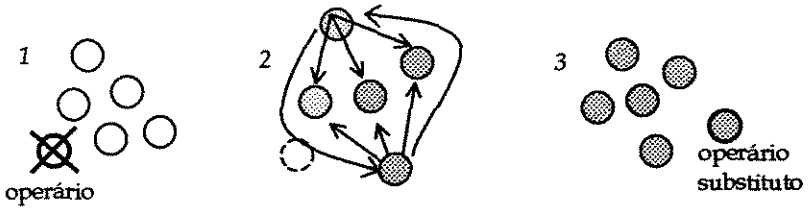


Figura 5.3 - Concorrência entre coordenadores.

1. O operário falha.
2. Dois agentes iniciam a recuperação simultaneamente.
3. Apenas o coordenador que obtiver $\left\lceil \frac{n+1}{2} \right\rceil$ de votos, terá sua lista aprovada.

Exemplo 4 - **Falha de comunicação**

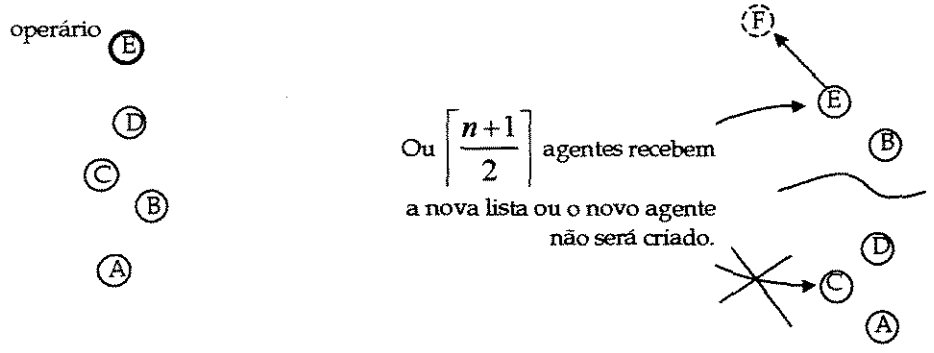


Figura 5.4 - Falha de comunicação impede a criação do novo operário.

O grupo de cinco agentes está pronto para iniciar um novo passo. O operário `E` torna-se o coordenador e propõe uma nova lista; há quorum. Ocorre uma falha de comunicação e os agentes `C`, `D` e `A` não recebem a nova lista, conseqüentemente, não respondem "confirmado". O novo operário `F` não é criado. Ou $\left\lceil \frac{n+1}{2} \right\rceil$ dos agentes da última lista válida confirmam o recebimento da nova lista ou o novo agente não é criado. Neste exemplo, `E`, `B` conhecem uma lista inválida com referência a um operário `F` que nunca existiu, o qual passa a ser visto como falho. Os agentes `E`, `B` tentarão recuperá-lo o que, de fato, representa uma nova tentativa de dar o passo.

Exemplo 5 – **Falha de comunicação quando utilizada uma versão do protocolo sem confirmação de recebimento de lista**

Se não houver garantia do recebimento da nova lista por pelo menos $\left\lceil \frac{n+1}{2} \right\rceil$ agentes da lista válida anterior, a propriedade da unicidade da execução estará comprometida. Na figura abaixo, um grupo de agentes ($n = 5$) executa (com falhas) uma versão do protocolo, para avançar de passo, que não exige a confirmação do recebimento da nova lista para criar os novos agentes. Neste exemplo, está sendo utilizada replicação linear em que o novo operário substitui o agente mais antigo; a nova lista é recebida apenas por dois agentes da antiga lista válida. Depois de criado o novo agente, é possível formar dois grupos que podem progredir independentemente.

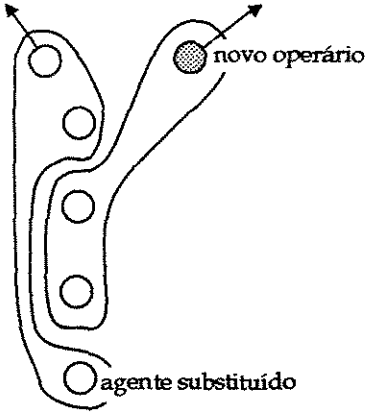


Figura 5.5 – É possível formar dois grupos de agentes independentes quando o novo agente é criado sem que o mínimo de $\left\lceil \frac{n+1}{2} \right\rceil$ agentes da última lista válida seja notificado.

Exemplo 6 – **Histórico de listas**

Um grupo de agentes ($n = 5$) era inicialmente integrado por A, B, C, D, E. A cada passo, o agente mais antigo é isolado em outra partição e não recebe a nova lista. Após cinco passos, tem-se a seguinte configuração:

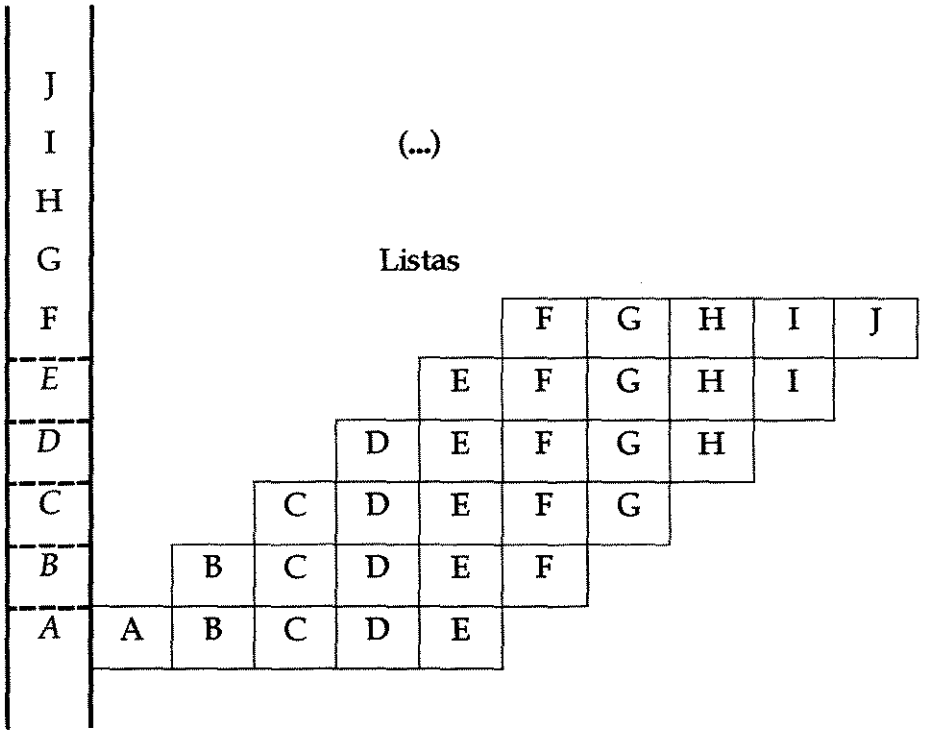


Figura 5.6 – Histórico de listas.

Vamos supor que após os cinco passos todos os agentes se reúnam em uma mesma partição e A torne-se coordenador. Ao contatar B, A descobre o agente F, depois descobre G e, finalmente, percebe que a lista {F, G, H, I, J} é válida porque está sendo utilizada por cinco agentes, que é um número maior que $\left\lceil \frac{n+1}{2} \right\rceil$. Logo A, B, C, D, E são inativos.

5.7 Correção

Nesta seção, é apresentada a argumentação que um grupo de agentes coordenado pelo protocolo da seção 5.5 executa completamente cada passo, uma única vez. Considere um número de cópias ideal igual a n.

***Lema 1*: A qualquer momento existem, em todo o sistema, no máximo n agentes que podem se combinar.**

Prova por indução:

Base: Existem n agentes definidos, na lista inicial (válida) `L`, que podem se combinar.

Hipótese (passo p_{i-1}): existem no máximo n agentes que podem se combinar, definidos por uma lista `L`, que é a lista válida com o maior número de versão.

Passo (passo p_i): Seja `L'` uma lista válida formada a partir de `L`, então $\left\lceil \frac{n+1}{2} \right\rceil$ agentes de `L` receberam `L'` e número de versão de `L'` > `L`. Vamos supor que, em algum momento um agente `A` de `L` que não está em `L'` tente se combinar. Pelo menos um agente dos $\left\lceil \frac{n+1}{2} \right\rceil$ que `A` contatar estará em `L'`, impedindo `A` de se combinar (informará a `A` da existência de `L'`).

Quando n for par e ocorrerem falhas de modo que `L'` seja distribuída apenas a $\frac{n}{2}$ agentes de `L`, somente n-1 agentes no sistema poderão se combinar. `A` não poderá se combinar porque não restará em `L` - `L'` $\left\lceil \frac{n+1}{2} \right\rceil$ agentes, entretanto `A'` ainda não terá sido criado. Este número voltará a ser n quando surgir a próxima lista válida (onde `A"` substituirá `A'` que nunca existiu).

Lema 2: **A qualquer momento, somente uma única lista poderá progredir.**

Inicialmente existe uma única lista no sistema. Para duas listas progredirem independentemente, em algum ponto da execução, uma lista L dá origem a duas listas válidas: L_1 e L_2. Nesse ponto seria necessário existirem no sistema $2\times\left\lceil\frac{n+1}{2}\right\rceil$ agentes que possam se combinar, o que é impossível. Conseqüentemente, o protocolo aqui apresentado impõe a propriedade da unicidade da execução.

Lema 3: **Na ausência de falhas o sistema progride.**

Para o sistema progredir é necessário existirem $\left\lceil\frac{n+1}{2}\right\rceil$ agentes na última lista válida que possam se comunicar, condição que é satisfeita na configuração inicial e nas subseqüentes pois n se mantém constante. Por hipótese não há empecilhos para comunicação entre agentes.

5.8 Extensão

O protocolo pode ser utilizado com listas de tamanhos diferentes. É apenas necessário respeitar a exigência de distribuir a nova lista a $\left\lceil\frac{n+1}{2}\right\rceil$ agentes da última lista válida.

Exemplo:

Seja a lista válida L com $n = 5$. Será preciso distribuir a nova lista L' para um mínimo de 3 agentes definidos em L. O tamanho de L' é 10. Para criar L" será necessário distribuir L" a pelo menos 6 agentes de L'.

5.9 Gerência do estado do operário

As seções anteriores discutiram um protocolo para recuperar um agente sem discutir como preservar sua informação. A falha de parada do agente operário

pode se limitar ao próprio agente ou ser provocada pela parada da agência onde executava. No segundo caso, será preciso que o substituto encontre um caminho alternativo, uma vez que a agência inicialmente selecionada estará indisponível. Mesmo não sendo provável poder repetir exatamente as mesmas operações do agente destruído é interessante que o substituto possua o estado mais recente. Para isso é possível, no início de cada passo:

- realizar *checkpoint*;
- enviar o estado do operário aos demais agentes do grupo.

A segunda opção torna-se custosa à medida que aumenta o tamanho do agente, sendo apropriado aplicar replicação com testemunhas [AM95] para reduzir a quantidade de informação transmitida. Neste esquema alguns membros recebem o estado, enquanto outros (testemunhas) participam apenas da votação.

5.10 Análise de Complexidade

Em relação ao tempo, o protocolo é completado em três *rounds* na ausência de falhas e `f`+3 *rounds* quando ocorrerem `f` falhas.

Na ausência de falhas, são utilizados:

- n-1 convites totalizando (n-1) × ($\log_2$(Id) + $\log_2$(número de versão)) bits (*Fase I*);

- n-1 mensagens "Sim" (*Fase II*);

- a nova lista com n posições é enviada a n-1 agentes, totalizando $\log_2$(n)×(n-1) *bits* (*Fase III*).

Capítulo 6 – Conclusão

Um sistema de agentes móveis tolerantes a falhas deve apresentar conceitos de unicidade e continuidade da execução [RS98], que podem ser alcançados com a utilização de replicação e um protocolo para a coordenação das réplicas, respectivamente. O controle sobre as réplicas pode ser feito passo-a-passo (ex: [JMS+98], [Ple02], protocolo do capítulo 5) ou ao final da execução [MTP00]. Realizar o controle passo-a-passo apresenta a vantagem de evitar que os recursos do sistema permaneçam bloqueados durante todo o período da execução, a fim de garantir isolamento das operações. O sistema também não é sobrecarregado com um grande número de agentes que executam operações redundantes as quais serão, posteriormente, descartadas.

Em [Ple02], é formalizada a coordenação do grupo de agentes como um problema de consenso em que cada passo, devem ser definidos:

- as agências que farão parte do próximo passo;
- o operário do próximo passo;
- o estado antigo.

O problema de consenso exige que dois agentes não decidam por valores diferentes; na sua terminação forte é permitido que somente os não falhos

decidam. Caso os agentes falhos possam se recuperar, não devem iniciar uma nova execução do protocolo, mas passar a utilizar o resultado que perderam; em [Ple02] é utilizado o protocolo de R-broadcast [CT96] (onde uma mensagem é enviada n^2 vezes) em conjunto com *log* para resolver esse problema.

Este trabalho apresentou um protocolo (capítulo 5) inspirado no trabalho de Chang e Maxemchuk [CM84] e apesar de se adequar a um modelo de tolerância a falhas igualmente amplo dispensa o R-broadcast e o uso de *log*. O protocolo do capítulo 5 utiliza uma definição mais flexível para o problema de coordenação dos agentes:

A qualquer momento, o número total de agentes no sistema que podem se combinar, é constante.

Existem três estratégias para construções de protocolos a partir desta definição:

1. Definir os agentes que podem se combinar. O protocolo do capítulo 5 utiliza esta estratégia definindo os agentes que podem se combinar na última lista válida.

   Outra possibilidade é interpretar a lista de cada agente como uma linha de um grafo dirigido. É possível controlar o número de agentes que podem se combinar, estabelecendo-se que dois agentes poderão se combinar somente se fizerem parte de um grafo completo de $\left\lceil \frac{n+1}{2} \right\rceil$ nós (um grafo completo é definido como um grafo não dirigido, portanto dois agentes têm uma aresta em comum nesse grafo se e somente se A conhece B e B conhece A).

2. Definir os agentes que não podem se combinar. Isso pode ser realizado através de uma lista de agentes falhos. É possível, por exemplo, estender o protocolo apresentado em [JMS+98] para um modelo com falha de partição, se durante a execução os agentes que não participarem forem incluídos em um *folder* de falhos e for utilizado o conceito de maioria. Suponha que um

grupo de cinco agentes se divida em duas partições contendo dois e três agentes. Quando for dado um passo (na partição com três agentes) estes receberão a nova lista que discriminará os agentes falhos que permanecerão impedidos de se reunir. Desse modo, o número de agentes do sistema será seis, entretanto o número máximo de agentes que podem se combinar, será quatro. A desvantagem da utilização da lista de falhos é excluir em caráter definitivo tais agentes.

3. Controlar o número total de agentes no sistema aumentando o valor da maioria de uma unidade para cada agente que estiver falho. Esta estratégia baseia-se na suposição de que qualquer falha tem natureza temporária, portanto chegará um momento em que um agente falho se recuperará e tentará se reunir novamente. Neste momento, porém, o grupo de agentes já terá dado um passo e, possivelmente, o referido agente fora escolhido para ser substituído, isto é, um novo agente terá sido criado sem a garantia de que outro tenha realmente substituído (e de fato não foi). É necessário aumentar o valor da maioria de uma unidade para compensar a possibilidade do número de agentes que podem se combinar, ter aumentado, impedindo que os agentes falhos formem uma maioria em outra partição. Se um agente isolado voltar a entrar em contato e confirmar sua destruição, o valor da maioria poderá ser reduzido da unidade. A desvantagem de tal abordagem é uma falha de parada incrementar permanentemente o valor da maioria, conseqüentemente o sistema tolera $\left\lfloor \frac{n}{2} \right\rfloor$ falhas de parada no total.

Todos os sistemas de agentes que realizam controle passo-a-passo, possuem um número total de agentes no sistema que podem se combinar constante a qualquer momento, mesmo que tenham definido o problema da coordenação dos agentes de maneira diferente.

[JMS+98] satisfaz a propriedade da unicidade da execução porque não há falhas de comunicação e quando um agente "desaparece" é porque parou.

Em [RS98], quando não é possível remover os agentes necessários, a execução bloqueia (*commit* de duas fases).

Em [AS99], [Ple02] apesar de exigir que apenas uma maioria de agentes decida por um mesmo valor, o resultado fica gravado em disco (DTD e *log* respectivamente) para atualizar os agentes que perderam o resultado, caso venham a se recuperar.

Uma possível extensão do presente trabalho é desenvolver e aplicar um *benchmark* para avaliar os sistemas de agentes. Este trabalho envolveria implementar os modelos discorridos acima em uma mesma linguagem para realizar comparações.

Referências Bibliográficas

[AK86] R. O. Anido e Kramer, J., "Synchronized por *avanço* and por r*etrocesso* recovery for communicating processes", 7th DCCS, Alemanha, Out. 1986.

[AL97] Adya, A., Liskov, B., "Lazy Consistency Using Loosely Synchronized Clocks", em Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODS), pp. 73-82, EUA, Aug 1997.

[AM95] R. O. Anido e Mendonça, N., C., "Protocols for maintaining consistency of replicated data.", relatório técnico *DCC-95-05*, Instituto de Computação da Universidade Estadual de Campinas, Campinas, Jun. 1995.

[AS99] Silva, F. M. de A., "A Transaction Model based on Mobile Agents", Informatik der Technischen Universität Berlin, Tese de Doutorado, 1999.

[CCS00] "Concurrency Control Service Specification", http://www.omg.org, versão 1.0, Abr. 2000.

[Con98] "Concórdia - Technology At A Glance", www.merl.com/projects/concordia/ WWW/Concordia-at-a-glance.pdf.

[CM84] Chang, J.M., Maxemchuk, N. F., "Reliable broadcast protocols", ACM Transactions on Computer Systems (TOCS), pag. 251-273, 1984.

[CPE97] Clements, P. E., Papaioannou, T., Edwards J., "Aglets: Enabling the Virtual Enterprise", Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement (ME-SELA '97), pag. 425.

[CT96] Chandra, T. D., Toueg, S., "Unreliable failure detectors for reliable distributed systems", J ACM, 42(2), pag. 225-267, Mar 1996.

[DHB97] Dunham, M. H., Hedal, A., Balakrishnan, S., "A mobile transaction model that captures both data and movement behavior", Mobile Networks and Applications 2, pag. 149-162, 1997.

[GR93] Gray, J., Reuter, A., "Transaction processing: concepts and techniques", 1993.

[GM95] Gosling, J., McGilton, H., "The Java Language Environment: a White Paper", Sun Microsystems inc., http://java.sun.com/whitePaper/java-whitepaper-1.html, 1995.

[GMS87] Garcia-Molina, H., Salem, K. "Sagas", relatório técnico CS-TR-070-87, Department of Computer Science, Princeton University, Jan. 1987.

[JMS+98] Johansen, D., Marzullo, K., Jacobsen, K., Schneider, F. B., "*NAP*: Practical Fault-Tolerance for Itinerant Computations", relatório técnico TR98-1716, CS Dept., Cornell University, Ithaca, Nova York, E.U.A., Nov. 1998.

[KR88] Klein, J., Reuter, A., "Migrating Trasactions", IEEE Workshop on the Future Trends of Distributed Computing Systems, Hong Kong, Sep. 1988.

[LA90] Lee, P. A. e Anderson, T., "Fault Tolerance: Principles and Practice", Springer-Verlag, 2ed., 1990.

[MF98] "Mobile Agent System Interoperability Facilities Specification", ftp://ftp.omg.org/pub/docs/orbos/98-03-09.pdf, 1998.

[Mos81] Moss, J. E. B., "Nested transactions: an approach to reliable distributed computing", PhD Thesis, relatório técnico MIT/LCS/TR-260, MIT Laboratory for Computing Science, Jun. 1981.

[MPT00] Mohindra, A., Purakayastha, A., Thati, P., "Exploiting non-determinism for reliability of mobile agents systems", International Conference on Dependable Systems and Networks (DSN'00), pag. 144-153, Nova York, Jun. 2000.

[PA86] Pâris, J. F., "Voting with witness: A consistency scheme for replicated files", Proc. of the 6th International Conference on Distributed Computer Systems, IEEE, pp.606-612, 1986.

[PB95] Pitoura, E., Bhargava, B., "Building Information Systems for Mobile Environments", 3rd International Conference on Information and Knowledge Management, Gaithesburg, MD, pag. 371-378, Nov. 1994.

[PKH98] Pu, C., Kaiser, G. E., Hutchinson, N., "Split-Transactions for Open-Ended Activities", 14th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos {CA}), Bancilhon and DeWitt (Eds), Los Angeles", 1988.

[PS00] Pleisch, S., Schiper, A., "Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems", 19th Symposium on Reliable Distributed Systems, IEEE Computer Society Press, 2000.

[PS01] Pleisch, S., Schiper, A., "FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach", Int'l Conference on Dependable Systems and Networks, IEEE Computer Society Press, 2001.

[Ple02] Pleisch, S., "Fault-Tolerant and Transactional Mobile Agent Execution", Tese de Doutorado, nº 2654, École Polytechnique fédérale de Lausanne, 2002.

[RS98] Rothermel, K., Straβer, M., "A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents", IEEE 17th Symposium on Reliable Distributed Systems, p. 100-108, 1998.

[SDR95] Schneider, F. B., Dag, J., Renesse, R. van, , "Operating System Support for Mobile Agents", 5th workshop on Hot Topics in Operating Systems - HOTOS, pag. 42-45, 1995.

[SK97] Silva, F. M. de A., Krause, S., "A Distributed Transaction Model Based on Mobile Agents", First International Workshop on Mobile Agents 97 (MA'97), Springer Lecture Notes in Computer Science 1219, 1997.

[TM00] Tripathi, A. R., Miller, R., "An Exception Handling Model for a Mobile Agent System", Exception Handling in Object-Oriented Systems, Sophia Antipolis – França, Jun. 2000.

[TS01] "Transaction Service Specification", versão 1.2, http://www.omg.org, Maio 2001.

[VB98] Vogle, H., Buchmann, A., "Using Multiple Mobile Agents for Distributed Transactions", Third International Conference of Cooperative Information Systems (COOPIS), http://www.computer.org/proceedings/coopis/8380/83800114abs.htm, 1998.

[Vei90] Veijalainen, J., "Transaction Concepts in Autonomous Database Environments", tese de doutorado, R.Oldenbourg Verlag, 1990.

[VKM97] Vogler, H., Kunkelmann, T., Moschgath, M. L., "An Approach for Mobile Agent Security and Fault Tolerance using Distributed Transactions", IEEE International Conference on Parallel and Distributed Systems – *(ICPADS'97)*, pag. 268-274. http://www.computer.org/proceedings/icpads/8227/82270268abs.htm, Seul, Coréia, Dez 1997.

[VOY] http://www.objectspace.com/products/voyager

[Whi96] White, J., "Mobile Agents White Paper", http://www.iiuf.unifr.ch/~chantem/white_whitepaper/whitepaper.html

[WR92] Wächter, H., Reuter, A., "The ConTract Model", [El92], Cap. 7, pag.219-263, 1992.